

- ◎ 传播先进思想 树立正确理念 全力打造非同一般的C语言参悟之旅
- ◎ 凭借深入浅出 旁征博引的讲述方式引领您进入C语言的多彩世界
- ◎ 通过新颖别致 妙趣横生的实例问题激发探究编程奥秘的无限乐趣

C语言 参悟之旅

左 飞 李召恒 编著



中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

C语言

参悟之旅

推荐语

大象无形、大方无隅——很难让人相信本书的作者们居然动手写了这样一本C语言的入门书。本以为或许作者们会写些很高深的东西出来，但当我看到书稿之后，却意识到他们将干瘪的语法描述得如此生动而通俗，或许这才是本书的价值所在，是他们所努力追求的方向。

本书作者们将丰富的开发经验凝聚在方寸之间，以轻松易懂、深入浅出出的笔调将C语言的世界描绘得有声有色。于是，让我有理由相信无论本书的读者处于何种阶段，都必然会从中有所收获，有所感悟。

真心地期待有更多的读者能够借助本书叩开C语言世界的大门，并从中发现编程的乐趣！

孟 坤

神州数码软件有限公司架构师

非常高兴看到作者完成了本书的创作，相信他们的努力付出一定不会令您失望。在我看来，本书无论从体例设置、还是描述方法上都有一定的独到之处。这些亮点一方面基于作者通过长期实践而累积的宝贵经验，另一方面则出于他们严谨求实的态度。现在，他们将自我所学倾囊相授，相信必然能惠及众人。

侯世超

北京雅普兰科技发展有限公司架构师

参悟

责任编辑：苏 茜 封面设计：九天科技 封面制作：白 雪 上架建议：计算机/程序设计/C语言



中国铁道出版社 计算机图书批销部
地址：北京市宣武区右安门西街8号
邮编：100054

网址：<http://www.tqbooks.net>
读者热线电话：(010) 63560056
销售服务电话：(010) 83550290/91 83550580



ISBN 978-7-113-10872-4/TP·3701 定价：49.00 元

专家荐序

介绍 C 语言的书已经有很多，为什么我要推荐这本书？

当我第一次看到书稿的时候，就为其中深入浅出的描述方式和独到的见解所吸引。这本书给人一种耳目一新的感觉，新颖的描述方式使其能够在众多同类书籍中别具一格。不难看出，作者的确下了一番功夫。

C 语言是一门应用极广的程序设计语言，拥有庞大的使用人群。它以简洁的语法形式，强劲高效的底层操作能力赢得了广大开发人员的青睐，然而，C 语言易学难精。长久以来，C 语言的某些特性（例如指针）总让人望而生畏。对于初学者来说，要想深入掌握 C 语言编程方法并不是一件容易的事。

作者在书名上冠以“参悟”二字，我想是经过反复推敲的，也足以反映出作者对 C 语言本质认识之深邃。书中从军队的组织结构引出数组的概念，以老鹰抓小鸡的游戏形式比拟线性链表，以《水浒传》中一百零八将的姓名、绰号和所使兵器等内容引入结构体，作者从一个现实已知的世界将读者逐渐引入到一个抽象的世界，慢慢参悟 C 程序设计的奥秘。

“师者，所以传道、受业、解惑也。”我想，《C 语言参悟之旅》的作者，正是出于这样的考虑，以传 C 语言编程之道，授软件研发之业，解初学者之惑为宗旨，以扎实的基础理论和丰富的开发经验为基础，从生活中常见的事物出发，以通俗易懂的言语和图文并茂的风格，引领 C 语言爱好者在 C 语言的王国里赏景游园，参究哲理，悟解精髓。我相信，藉用该书，会有更多的编程爱好者从初学走向熟练，从平凡走向精通。我们期待着有更多的读者掌握 C 语言程序设计的方法、技巧和真谛，从中体会到编程开发的乐趣与喜悦。

伟大的时代使青年人的作品彰显出时代的活力，衷心希望本书作者继续努力，沿着这条路走下去，创作出更多精品。

吴广茂
西北工业大学博士生导师、研究员



编者自序

——谁都有追逐梦想的权利

昨天在街上走，忽然有人拍了我一下。

原来是从前在单位聊过天的一个小保安。

那时，我大学刚毕业，分到单位不久。小保安从我初到单位实习的时候就遇见了。当时下班之后我经常待在办公室晚走一些，每次下楼都能碰到这个家伙在看 C++ 或者 Java 的书，而且看书的只有这个家伙，呵呵。

后来他一直是保安，而我一直在公司继续重复我原本的工作。每每在走廊、地铁站、楼梯间里面碰见他背个大包，我们都要相视一笑。

“又下课了？”

“是啊，呵呵。”

这次碰见我这么主动，感觉怪怪的。

“我辞职了。”

啊？搞笑吧，金融危机下还有人敢辞职啊？

“你去哪儿啊？”

“应聘到一家用友软件的合作商那儿了。”

恭喜了半天，感觉确实很欣慰。

我和他认识的一年半时间里，只是见过短短的几面，偶尔他借借我的 U 盘，或者让我帮他打印个身份证什么的。相比于别人，跟他认识的时间似乎只有几小时的样子。

一叶知秋，不经意间，我们也觉不到原来身边的树木都已经多了一圈年轮。不经意间，身边一个不让人留意的人已经华丽转身，飞走了。

不管怎么说，小保安用了一年的时间实现了一次人生的飞跃。静静的累积，静静的蜕变。比起别的东西，他的离开倒是我生活中不小的一个变化，更对比出自己的停滞。

也许是羡慕吧，羡慕那种有了目标默默的努力，然后修成正果的样子。

这也不禁让我想起了因为选秀节目“英国达人”而一炮走红的 Susan 大妈。其貌不扬（或

者可以夸张地说有点长相“雷人”)、打扮老土、身材发福、现年 47 岁的英国大妈苏珊·博伊尔,凭借一首音乐剧《悲惨世界》中的曲目《I dreamed a dream》(我曾有梦想)不仅赢得了评委和在场观众雷鸣般的掌声,更打动了无数人的心。

就在 Susan 大妈演唱之前,评委西蒙漫不经心地问道:“你的梦想是什么?” Susan 回答道:“做专业歌手,成为伊莲·佩吉(注:伊莲·佩吉是英国音乐剧第一夫人,曾享誉伦敦舞台界 30 年,堪称当代英国首席音乐剧红伶)那样的歌星。”

Susan 大妈努力地追逐着自己的梦想,如同她的歌曲一样——我曾有梦想。谁不曾有过年少轻狂的时光,谁不曾有过激动人心的梦想。看来人老不老,区别更多的还是在心里面。即使 Susan 大妈已经有 47 岁了,但勇于追逐梦想的她依然年轻!

想想我们,也许在大学里面有太多的经历,也许是小保安忽然唤醒了我的这么多想法,然后让我忽然意识到付出总会有回报。

也许……

有的人,从头放弃了,

有的人,没有相信过,

有的人,途中懈怠了……

也许就没有什么也许,失败者有着不同的失败经历,成功者有着他们的不同梦想。

让人感动的是,这些人对梦想执着地追求,而让人思量的是,社会让他们实现了梦想。

每个人都有追逐梦想的权利,只是是否把握得住,要看他们自己是否努力去追求了。

感慨了这么多,并写在我新书的最前面,希望能把这本书奉献给所有曾有梦想或正在追逐梦想的人们,真心希望所有的人能够梦想成真!

2009 年 10 月



前言

为什么还学 C 语言

C 语言不仅是理工科学生必修的课程，同时也是国家计算机等级考试的必考科目。时至今日，仍然有许多软件系统采用 C 语言来作为开发语言。C 语言具有语法简洁、灵活方便，支持底层操作、开发自由度大，可移植性好、执行效率高等诸多优点。

学好 C 语言不仅能够为程序设计本身打开一扇天窗，更能够帮助初学者实现举一反三、触类旁通的愿望。以 C 语言为基础纵向可以继续学习数据结构与算法方面的相关知识，横向可以为继续学习面向对象的 C++ 或 Java 语言奠定基础。可见 C 语言不失为计算机程序设计入门的首选语言。

初学者的困惑

尽管学习 C 语言是一件令人向往的事情，但是很多初学者在深入学习之后往往感到困惑：

- ❑ 对新的概念感到陌生，理解起来不够透彻；
- ❑ 面对复杂难懂的语法规则往往是学过之后，印象不深，写起代码又感到不知所措，无从下手。

为了帮助读者全面系统地掌握 C 程序设计的要领，帮助那些在门口徘徊却迟迟无法进入状态的读者扫清障碍、冲破险阻，我们编写了这本《C 语言参悟之旅》。在这本书里笔者总结了以往学习和实践 C 程序设计的经验，运用轻松的笔调和有趣的实例引领读者步步深入，探索 C 语言的奥秘。

本书特点

目前，市场上可见的 C 语言著作可谓是琳琅满目、层出不穷。其中也不乏经典之作，然而时过境迁，很多写于 20 世纪八九十年代的 C 语言书籍已经跟不上快速变换的时代步伐。为了适应新一代 C 程序爱好者的需求，我们重拾了 C 语言这个传统的话题，希望能够写出一些更加实用、有特色的内容。

C 语言作为一种“活”的语言，它在不断充实和进步，我们希望本书能够不断适应时代的变换和读者的需求。为此，本书力求突出以下几个特点：

☑ 强调概念阐释的通俗性

为了让所有读者都能够在最初接触一门语言时就做到准确深刻地理解每一个概念，而不至于留下“后遗症”，本书不但配有大量插图来帮助读者理解，还为绝大部分关键概念设计了形象的比喻或日常生活参照，以便帮助读者从实际生活中理解这些关键概念。

☑ 强调编程实践的先导性

纸上得来终觉浅，深知此事需躬行。学习编程绝对不能纸上谈兵，丰富的编程实践不仅能够帮助读者提升实战能力，而且可以帮助读者对于相关知识点的深化理解和掌握。本书在编程实例的设计上经过反复推敲，选定了比较恰当的实例。与其他 C 语言书籍不同的是，这些实例一方面是一些实际工程问题的缩影，另一方面缘自一些经典算法问题，具有较强的实际意义和参考价值。实例背景生动丰富，便于引导读者进入情景，产生阅读快感。

☑ 强调先进思想的重要性

本书在撰写时无论是对实例的设置，还是概念的描述，都力求帮助读者树立正确的、先进的编程思想。这主要包括结构化的程序设计思想、良好的编码风格与正确的程序书写规范，还包括一些结构化数据类型的组织及算法设计思想。这些思想的树立能够全方位地帮助读者提高编程水平，达到量变到质变的效果。

☑ 强调深入学习的后续性

学习 C 语言仅仅是叩开了编程世界的大门，日后仍然任重而道远。为了能够使读者继续深入的学习，不断的进步，本书在数据结构和算法设计方面做了大量的铺垫，将相关知识融入于 C 语言学习之中，方便读者在完成 C 语言相关内容的学习之后继续学习数据结构方面的知识，为读者实现平滑过度披荆斩棘、保驾护航。

本书内容

全书共分 11 章，系统详尽地介绍了 C 语言程序设计的基本方法，主要包括程序设计与 C 语言概述，数据及数据类型，运算符、表达式和语句，流程控制，函数，数组与字符串，指针，预处理，结构体与共用体，文件以及动态数据结构等内容。

全书内容丰富、结构清晰、实例代码详尽，介绍经典算法、经典问题和大量的示例程序，配有很多相关的插图，具有很强的参考意义。

本书中的所有程序都在 Visual C++ 6.0 环境下调试通过，完整的代码可从本书的相关网站下载。欢迎广大读者访问编者的博客 <http://blog.csdn.net/baimafujinji>，与编者就 C 语言程序设计的相关问题进行讨论。

本书阅读对象

本书适合作为 C 语言初学者的入门教材，尤其适合自学；也可作为大专院校在校师生相关课程的参考书及从事 C 语言开发的程序员的参考手册。

本书编者

本书由左飞、李召恒编著。中国航空工业西安航空计算技术研究所原总工程师、陕西省软件行业协会常务理事、西北工业大学博士生导师吴广茂研究员欣然为本书作序推荐，在此向吴老师表示最诚挚的谢意。西安交通大学硕士研究生潘聪、西北工业大学博士生黄丽江、西安石油大学王晓燕老师、长安大学硕士研究生徐波等在本书的创作过程中给予了很大的帮助，在此一并表示衷心的感谢。

问题解答途径

本书由于创作时间仓促，纰漏和欠缺之处在所难免，希望读者不吝赐教和批评。联系信箱：li_zhaoheng@126.com 和 ww_bei_jing@yahoo.com.cn。

编 者

于 2009 年 10 月



目 录

第 1 章 程序设计与 C 语言概述	1
1.1 计算机程序	2
1.1.1 什么是程序	2
1.1.2 什么是计算机程序	3
1.1.3 程序设计	4
1.2 计算机语言	4
1.2.1 语言	4
1.2.2 什么是计算机语言	4
1.2.3 计算机语言简史	4
1.2.4 高级语言的执行方式	6
1.3 C 语言概述	7
1.3.1 为什么叫“C 语言”	7
1.3.2 C 语言的版本	7
1.3.3 C 语言的特点	7
1.3.4 C 语言的应用	9
1.4 第一个 C 程序	9
1.4.1 为什么选“Hello, World”	9
1.4.2 “Hello, World” 程序	10
1.4.3 “Hello, World” 程序解析	10
1.4.4 C 程序结构特点解析	10
1.4.5 C 程序是如何执行的	12
1.5 Microsoft Visual C++ 6.0 集成开发环境简介	13
1.5.1 集成开发环境 (IDE)	13
1.5.2 集成开发环境的功能	13
1.5.3 为什么选择 Microsoft Visual C++ 6.0	14
1.5.4 Microsoft Visual C++ 6.0 的版本	15
1.5.5 Microsoft Visual C++ 6.0 的安装	15
1.5.6 项目和工作区	18
1.5.7 Visual C++ 6.0 界面简介	19
1.5.8 常用菜单项	21
1.5.9 常用工具栏	22
1.5.10 视图窗格简介	23



1.5.11 代码颜色.....	26
1.5.12 使用 Visual C++ 6.0 编写和运行 “Hello, World” 程序	26
第 2 章 数据及数据类型.....	30
2.1 数据在计算机中的表示.....	31
2.1.1 数据	31
2.1.2 字符集和标识符.....	31
2.1.3 数据在计算机中的表示.....	33
2.2 数据类型.....	34
2.2.1 数据类型的解释.....	35
2.2.2 C 语言中的数据类型	35
2.3 常 量.....	36
2.3.1 整型常量.....	36
2.3.2 实型常量.....	38
2.3.3 字符常量.....	39
2.3.4 字符串常量.....	40
2.3.5 符号常量.....	41
2.4 变 量.....	42
2.4.1 给变量命名.....	42
2.4.2 变量定义.....	42
2.4.3 变量名与变量的值.....	43
2.4.4 变量初始化.....	44
2.4.5 赋值	44
2.4.6 整型变量.....	45
2.4.7 实型变量.....	48
2.4.8 字符变量	50
2.5 数据的输入/输出.....	50
2.5.1 什么是输入/输出	51
2.5.2 C 语言中输入/输出的实现	51
2.5.3 格式化输出——printf() 函数	51
2.5.4 格式化输入——scanf() 函数.....	56
2.5.5 字符的输出——putchar() 函数	60
2.5.6 字符的输入——getchar() 函数	61
2.5.7 输出字符串——puts() 函数	61
2.5.8 读取字符串——gets() 函数.....	62
第 3 章 运算符、表达式和语句	64
3.1 运算符和表达式概述.....	65
3.1.1 运算符	65
3.1.2 表达式	65

3.1.3	运算符的优先级和结合性.....	66
3.2	算术运算符与算术表达式.....	67
3.2.1	算术运算符.....	67
3.2.2	算术表达式.....	68
3.2.3	算术表达式的求值.....	68
3.3	赋值运算符与赋值表达式.....	69
3.3.1	简单赋值.....	69
3.3.2	左值和右值.....	70
3.3.3	复合赋值.....	71
3.3.4	赋值运算符的副作用.....	71
3.3.5	子表达式的求值顺序.....	72
3.4	类型转换.....	72
3.4.1	类型转换简述.....	72
3.4.2	自动类型转换.....	73
3.4.3	强制类型转换.....	76
3.5	自增和自减运算符.....	78
3.5.1	简化特殊的运算符.....	78
3.5.2	使用自增和自减运算符注意事项.....	79
3.6	逗号运算符与逗号表达式.....	79
3.6.1	逗号表达式.....	80
3.6.2	使用逗号表达式注意事项.....	80
3.7	关系运算符与关系表达式.....	81
3.7.1	关系运算符.....	81
3.7.2	关系表达式的值.....	81
3.7.3	使用关系运算符注意事项.....	81
3.8	逻辑运算符与逻辑表达式.....	82
3.8.1	逻辑运算符.....	82
3.8.2	逻辑表达式.....	82
3.8.3	“短路”计算.....	83
3.9	位运算符.....	83
3.9.1	C 语言的位运算符.....	83
3.9.2	按位与运算符.....	84
3.9.3	按位或运算符.....	85
3.9.4	按位异或运算符.....	85
3.9.5	按位取反运算符.....	86
3.9.6	左移运算符.....	87
3.9.7	右移运算符.....	88
3.9.8	位运算中的整数提升.....	88
3.9.9	位运算赋值运算符.....	89



3.10	sizeof 运算符	89
3.10.1	使用 sizeof	89
3.10.2	sizeof 的结果	90
3.10.3	sizeof 的优先级	91
3.10.4	各种类型数据长度的计算	91
3.11	语句	91
3.11.1	什么是语句	92
3.11.2	语句类型	92
3.11.3	赋值语句	93
第 4 章	流程控制	95
4.1	流程的表示方法	96
4.1.1	自然语言表示法	96
4.1.2	流程图表示法	96
4.2	顺序结构	97
4.2.1	什么是顺序结构	97
4.2.2	顺序结构程序设计方法	98
4.3	分支结构	100
4.3.1	什么是分支结构	100
4.3.2	if 语句的解释	101
4.3.3	if 语句的 3 种形式	101
4.3.4	嵌套的 if 语句	105
4.3.5	应用 if 语句注意事项	106
4.3.6	条件运算符的解释	109
4.3.7	应用条件运算符注意事项	110
4.3.8	switch 语句的解释	111
4.3.9	应用 switch 语句注意事项	113
4.3.10	分支结构程序设计方法	113
4.4	循环结构	116
4.4.1	什么是循环结构	116
4.4.2	关于 while 语句的解释	116
4.4.3	应用 while 语句注意事项	117
4.4.4	关于 do...while 语句的解释	118
4.4.5	应用 do...while 语句时防止死循环	119
4.4.6	不确定循环和计数循环	119
4.4.7	关于 for 语句的解释	120
4.4.8	使用 for 语句注意事项	121
4.4.9	选择哪种循环实现方式	123
4.4.10	循环中的循环	124
4.4.11	循环结构程序设计方法	125

4.5 跳转结构.....	128
4.5.1 什么是跳转结构.....	128
4.5.2 break 语句.....	129
4.5.3 continue 语句.....	131
4.5.4 goto 语句.....	132
4.5.5 C 语言中保留 goto 语句的原因.....	133
第 5 章 函数.....	136
5.1 函数与结构化程序设计.....	137
5.1.1 函数是“黑盒子”.....	137
5.1.2 数学函数与 C 语言函数.....	138
5.1.3 C 语言函数中的库函数.....	139
5.1.4 结构化的程序设计.....	139
5.2 函数的使用.....	140
5.2.1 函数的定义.....	140
5.2.2 函数的类型.....	141
5.2.3 函数的返回值.....	142
5.2.4 函数的参数.....	145
5.2.5 函数的调用.....	148
5.2.6 函数的嵌套——蒙特卡罗法求圆周率 π	151
5.3 递归.....	153
5.3.1 递归的定义.....	154
5.3.2 使用递归的原则.....	155
5.3.3 分治法与汉诺塔.....	162
5.3.4 回溯法与八皇后问题.....	164
5.4 变量的作用域.....	168
5.4.1 局部变量.....	168
5.4.2 局部变量的作用域.....	168
5.4.3 全局变量.....	169
5.5 变量的存储类型.....	171
5.5.1 auto 变量.....	172
5.5.2 static 局部变量.....	173
5.5.3 register 变量.....	174
5.5.4 extern 变量.....	176
5.5.5 static 外部变量.....	178
5.6 执行多文件程序.....	179
5.6.1 内部函数.....	179
5.6.2 外部函数.....	179
5.6.3 多文件程序实例.....	181



第 6 章 数组与字符串	183
6.1 一维数组的使用	184
6.1.1 数组概念的引入——中国古代军队编制	184
6.1.2 数组元素的使用	185
6.1.3 数组的初始化	185
6.1.4 小心访问越界	187
6.1.5 数组应用举例	187
6.2 数组类型的参数	190
6.2.1 以数组作为参数	190
6.2.2 避免数组被修改	191
6.2.3 函数返回数组的两种方法	192
6.3 多维数组的使用	195
6.3.1 从一维到二维	195
6.3.2 初始化及使用二维数组	195
6.3.3 多维数组应用举例	197
6.4 字符数组	201
6.4.1 定义与初始化	201
6.4.2 字符串的使用	201
6.4.3 字符串的处理——大小写转换函数	203
6.4.4 字符串的处理——字符串比较函数	203
6.4.5 字符串的处理——字符串长度的获得	204
6.4.6 字符串的处理——字符串连接函数	204
6.4.7 字符串的处理——字符串复制函数	205
6.4.8 字符串应用举例	207
第 7 章 指针	210
7.1 指针与地址	211
7.1.1 内存和地址的概念	211
7.1.2 定义指针变量	213
7.1.3 使用指针变量	214
7.2 指针与数组	217
7.2.1 用指针访问数组元素	217
7.2.2 直接插入排序	219
7.2.3 用指针操作多维数组	220
7.2.4 Z 字形编排过程	223
7.2.5 复杂指针运算的解析	225
7.3 使用字符串指针变量	226
7.3.1 指向字符串的指针	226
7.3.2 与字符数组的比较	230

7.3.3 如何输出其自身的程序.....	233
7.4 指针与函数.....	234
7.4.1 将指针用作函数参数.....	234
7.4.2 指向函数的指针.....	236
7.4.3 指针对于指令的访问是受限制的.....	238
7.4.4 使用指向函数指针的语法来实现编程.....	238
7.4.5 返回值为指针的函数.....	240
7.5 复合多维指针的使用.....	244
7.5.1 指针数组的使用.....	244
7.5.2 指向指针的指针.....	247
7.5.3 main ()函数的参数.....	249
7.5.4 main ()函数参数应用实例.....	249
第8章 预处理.....	252
8.1 预处理器概述.....	253
8.1.1 预处理器的生活方式.....	253
8.1.2 使用 Microsoft Visual C++ 6.0 生成预编译程序.....	254
8.1.3 预处理指示分类.....	256
8.1.4 预处理指示规则.....	256
8.2 宏定义.....	257
8.2.1 无参宏定义.....	257
8.2.2 带参宏定义.....	258
8.2.3 带参宏定义与函数.....	259
8.2.4 使用宏时注意事项.....	261
8.2.5 至关重要的圆括号.....	262
8.2.6 预定义宏.....	263
8.3 条件编译.....	264
8.3.1 条件编译的形式.....	264
8.3.2 条件编译的作用.....	267
8.4 文件包含.....	268
8.4.1 头文件.....	268
8.4.2 文件包含的形式.....	269
8.4.3 使用文件包含时注意事项.....	270
8.5 其他指示.....	271
8.5.1 #error 指示.....	271
8.5.2 #line 指示.....	271
8.5.3 #pragma 指示.....	272
8.6 “#”和“##”运算符.....	272
8.6.1 “#”运算符.....	272



8.6.2	“##”运算符	273
8.7	预处理实例	273
8.7.1	简单计算器程序	273
8.7.2	程序分析	276
8.7.3	程序中的预处理	277
第9章	结构体与共用体	279
9.1	结构体	280
9.1.1	什么是结构体	280
9.1.2	结构体实例——《水浒传》中的一百单八将	280
9.1.3	结构体类型与结构体变量	281
9.1.4	结构体变量的定义	282
9.1.5	定义结构体变量注意事项	283
9.1.6	结构体变量的初始化	283
9.1.7	结构体变量的引用	284
9.1.8	引用结构体变量注意事项	285
9.1.9	结构体数组	286
9.1.10	指向结构体的指针	289
9.1.11	结构体与函数	291
9.1.12	位域	298
9.2	共用体	301
9.2.1	什么是共用体	301
9.2.2	共用体与结构体	302
9.2.3	共用体变量的初始化	303
9.2.4	使用共用体注意事项	304
9.2.5	结构体和共用体综合实例——“梁山好汉的比武大会”	305
9.3	枚举	308
9.3.1	什么是枚举	308
9.3.2	枚举变量的定义与取值	309
9.3.3	“表里不一”的类型	310
9.3.4	枚举应用举例——“向你问好的程序”	311
9.4	用户自定义类型——typedef	312
9.4.1	什么是typedef	312
9.4.2	创建typedef简单方法	313
9.4.3	typedef和#define	313
9.4.4	typedef的两个重要作用	314
第10章	文件	316
10.1	理解文件的基本概念	317
10.1.1	什么是文件	317

10.1.2	什么是流.....	318
10.1.3	处理文件的方法.....	319
10.1.4	缓存	320
10.1.5	文本文件和二进制文件	321
10.2	文件的打开与关闭	321
10.2.1	文件类型指针.....	321
10.2.2	文件的打开.....	323
10.2.3	文件操作类型及应用	323
10.2.4	文件的关闭.....	325
10.3	文件的基本操作	327
10.3.1	文件中的字符读/写.....	327
10.3.2	按字符进行读/写文件——文件复制的功能	329
10.3.3	文件中字符串读/写.....	330
10.3.4	其他文件读/写函数.....	334
10.3.5	文件位置定位.....	334
10.3.6	数据块的读/写	336
10.4	处理二进制文件	338
10.5	文件缓冲区处理	341
10.5.1	文件缓冲区的清除	341
10.5.2	文件缓冲区的设置.....	343
10.6	文件操作的检测	345
第 11 章	动态数据结构	347
11.1	动态内存管理	348
11.1.1	为什么使用动态内存分配.....	348
11.1.2	如何实现动态内存管理.....	348
11.1.3	关于动态内存分配的说明	351
11.2	链表概述	352
11.2.1	单向链表与数组.....	353
11.2.2	单向链表——老鹰捉小鸡	353
11.2.3	链表存储方式优缺点.....	354
11.2.4	不同单向链表间的合并.....	354
11.3	链表的操作及实现	355
11.3.1	链表的建立.....	355
11.3.2	链表的遍历.....	358
11.3.3	链表结点的删除.....	359
11.3.4	链表结点的增加.....	361
11.3.5	结点删除函数中的 free()函数.....	363
11.3.6	链表的应用实例.....	363



11.4 栈	364
11.4.1 栈定义——散乱的盘子	364
11.4.2 栈的特点	365
11.4.3 栈工作原理	365
11.4.4 栈与链表	366
11.4.5 栈的应用举例——括号匹配问题	368
11.5 队列	369
11.5.1 队列的特点	369
11.5.2 队列定义——排队等待买票的人	370
11.5.3 队列应用	370
11.5.4 队列与栈的不同	370
11.5.5 队列创建	370
11.5.6 基于链表实现的队列——链式队列	371
11.5.7 队列的实现	372
附录 I C 语言运算符及其优先级汇总表	376
附录 II 标准 ASCII 码字符集	378
参考文献	380





第1章

程序设计与C语言概述



当今世界，计算机程序已经成为人们生活、学习和工作中必不可少的东西，其应用可以说是渗透到了人们生活的方方面面。而这些计算机程序都是用计算机语言来编写的，所以，要编写计算机程序首先要选择一门计算机语言。在众多的计算机语言中，C语言凭借其众多优良特性脱颖而出，成为一把编写计算机程序的“利器”。

本章就从计算机程序和计算机语言入手，对C语言的概念、特点、应用、结构和开发工具进行简要的介绍。





1.1 计算机程序

计算机程序，顾名思义就是运行在计算机上的程序，所以，要了解计算机程序首先要了解什么是程序。本节就从程序入手，介绍程序、计算机程序，以及程序设计。

1.1.1 什么是程序

提到程序一词，很多人可能会联想到计算机，认为程序来源于计算机。但并非如此，程序来源于生活，通常是指完成某些事务的一种方式 and 过程。从表面上看，可以将程序看成是对一连串动作的操作过程的表述。在日常生活中，有许多程序的实例，例如，小李要乘坐火车从西安到深圳的行为可以表述为：

第 1 步 小李购买到达深圳的车票；

第 2 步 小李按时到西安站乘车；

第 3 步 小李到达深圳。

此程序是一个最简单的程序，将其称为直线型程序。表述直线型程序只需给出一个包含其中各个基本步骤的序列即可，如图 1.1 所示。

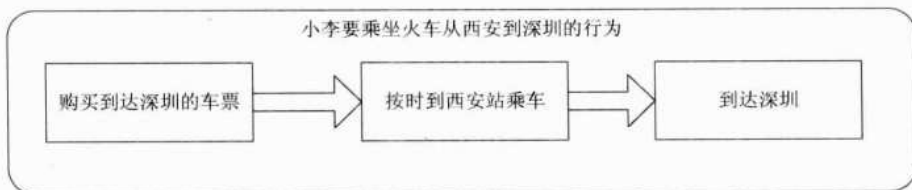


图1.1 小李事务序列图一

按此序列的顺序执行这些步骤，即可完成这些事务。那么，假设这些事务中，有些情况发生了变化，比如说现在从西安到深圳的车票卖完了，而只有西安到广州的车票，则此行为可以表述为：

第 1 步 小李购买到达广州的车票；

第 2 步 小李按时到西安站乘车；

第 3 步 小李乘车到达广州；

第 4 步 这时有两种选择：

a. 继续乘坐火车从广州到深圳；

b. 乘坐汽车或者其他交通工具从广州到深圳；

第 5 步 小李到达深圳。

此程序和前一个程序都是完成从西安到深圳这件事，但是因为情况发生了变化，所以此程序就要复杂一些。可以看到，此程序不再是一个平铺直叙的动作序列，其中的步骤更多，而且还出现了需要选择的行为和可能重复出现的动作，如图 1.2 所示。

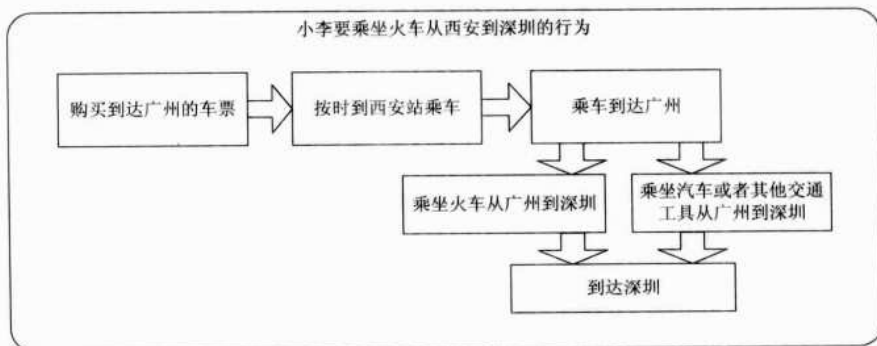


图1.2 小李事务序列图二

再来仔细研究一下上面的程序，可以看出，它还可以进一步细化。例如，在第4步后选择了步骤a，则又是一个买票、乘车、到达的程序。可见程序是可以拆分的，可以将一个复杂的程序拆分为若干个简单的程序，从而使问题变简单，如图1.3所示。

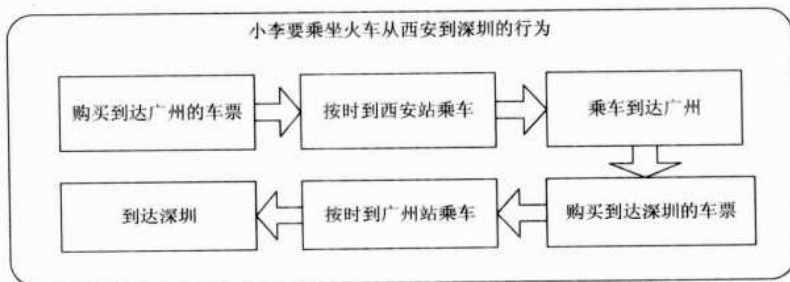


图1.3 小李事务序列图

一个程序总有开始和结束。在上面的例子中，买票是开始，到达深圳是结束。在执行程序的过程中，执行者（执行程序的人或者其他事物）需要从程序表述的开始位置开始，然后执行一系列的动作，当到达结束位置时事务就完成了，程序也就结束了。


1.1.2 什么是计算机程序

简单地说，计算机程序就是运行在计算机上的程序。具体来说，计算机程序是指能够使计算机做出信息处理行为并产生一定结果的指令或指令组合。其中指令就是能被计算机识别并执行的二进制代码，它规定了计算机能完成的某种操作。将这些指令按照某种顺序排列起来就构成了计算机程序，而执行计算机程序的过程就是计算机工作的过程。

因为计算机程序是一种程序，所以它拥有程序的一切特性。比如说，有简单和复杂之分，复杂的计算机程序可以拆分为若干个简单的计算机程序，总有开始和结束。

但是，计算机程序和日常生活中的程序并不完全相同。例如，人们的日常生活、工作中存在着许多变数，所以完成这些工作的程序也并非一成不变。许多事情并不需要完全按照程序来做，经常会根据现实情况灵活处理。而计算机对程序的执行则非常严格，会一步一步地按照计算机程序中的指令来执行，是一点“商量”的余地都没有的。



注意：本书后面章节中的程序都是计算机程序，为了介绍方便，后面的章节中统一用程序这个词来表示计算机程序。

1.1.3 程序设计

人们把编制程序的工作称为程序设计或者编程，编程工作的“产品”就是程序，从事程序设计的人员称为程序员。人们熟悉的一些软件系统，例如 Windows、Microsoft Office、QQ 等软件，都是由程序组成的。

计算机需要完成的工作就是计算，比如加、减、乘、除这些基本计算。不过计算机能完成的计算远远不止这些，所有对数据进行处理的过程都是计算，例如，数学计算、实验模拟、图形处理、音频处理、视频处理等。要完成多种多样复杂的计算，就必须依靠程序和程序设计，所以从计算机诞生开始就有了程序设计工作。

1.2 计算机语言

要进行程序设计工作，就必须使用计算机语言，而要了解计算机语言则必须了解语言。本节就从语言入手，介绍语言、计算机语言、计算机语言的发展史，以及高级语言的执行方式。

1.2.1 语言

对于语言，人们并不陌生，因为在人们的日常生活中，使用最多的就是语言。人类始终是语言的核心。在笔者看来，语言就是人们用于和其他人以及客观世界交流的一种工具。那么，当人类要和计算机进行交流时，就需要一种特殊的语言，那就是计算机语言。

1.2.2 什么是计算机语言

计算机语言是指用于人与计算机之间通信的语言，是人与计算机之间传递信息的媒介。因为它是用来进行程序设计的，所以又称程序设计语言或者编程语言。

计算机语言是一种特殊的语言。因为它是用于人与计算机之间传递信息的，所以人和计算机都能“读懂”。具体地说，一方面，人们要使用计算机语言指挥计算机完成某种特定工作，就必须对这种工作进行特殊描述，所以它能够被人们读懂。另一方面，计算机必须按计算机语言描述来行动，从而完成其描述的特定工作，所以能够被计算机“读懂”。

1.2.3 计算机语言简史

现代汉语中汉字的最初形式被认为是甲骨文。正如从甲骨文到现代汉字的演变过程中总是伴随着巨大的变化一样，计算机语言在诞生的短短几十年里，也经过了一个从低级到高级的演变过程。具体地说，它经历了机器语言、汇编语言、高级语言 3 个阶段。

1. 机器语言阶段

计算机所使用的是由“0”和“1”组成的二进制数，二进制数是计算机语言的基础。在计算机发展的早期，所谓的编程就是写出一连串由“0”和“1”组成的指令序列并交由计算机执行，这种编程语言就是机器语言，也是第一代计算机语言。一个典型的机器语言程序如下：



```
1010 1111
0011 0111
0111 0110
...
```

此种语言是针对特定型号计算机的，相当于直接和计算机进行交流，计算机不用转化就能执行，所以运算效率是所有计算机语言中最高的。但是，使用机器语言却是十分痛苦的，特别是在程序有错需要修改时，更是如此。而且，由于每台计算机的指令系统往往各不相同，所以，在一台计算机上执行的程序，要想在另一台计算机上执行，必须另编程序，相当麻烦。所以，使用机器语言不久后，很快就出现了第二代计算机语言——汇编语言。

2. 汇编语言阶段

为了减轻使用机器语言编程的痛苦，程序员对其进行了一种有益的改进：用一些简洁的英文字母、符号串来替代一个特定的指令的二进制串，比如，用“ADD”代表加法、“MOV”代表数据传递等，这种程序设计语言就称为汇编语言，也是第二代计算机语言。前面所示的机器语言程序用汇编语言表示如下：

```
MOV A, 47
ADD A, B
HALT
...
```

比较两段程序可以看出，人们更容易读懂并理解汇编语言编写的程序在干什么，程序的纠错及维护也都变得方便了许多。然而计算机是不认识这些符号的，这就需要一个专门的程序，负责将这些符号翻译成二进制数的机器语言，这种翻译程序被称为汇编程序。

汇编语言同样十分依赖于计算机硬件，移植性不好，但效率却很高，针对计算机特定硬件而编制的汇编语言程序，能准确发挥计算机硬件的功能和特长，程序精练且质量很好，所以至今仍是一种常用的程序设计语言。

汇编语言仍然是“面向机器”的语言，也就是说它是针对某种特定机器编写的，且仍然很难懂。在编写一些大型的程序时，它已经不能很好地满足要求，所以又出现了第三代计算机语言——高级语言。

3. 高级语言阶段

从最初与计算机交流的痛苦经历中，人们意识到，应该设计一种这样的语言，接近于数学语言或人类的自然语言，同时又不依赖于计算机硬件，编出的程序能在所有机器上使用。经过努力，1954年，第一个完全脱离计算机硬件的高级语言——FORTRAN问世。

50多年来，共有几百种高级语言出现，其中有重要意义的有几十种，影响较大、使用较广泛的有FORTRAN、ALGOL、COBOL、Basic、LISP、SNOBOL、PL/1、Pascal、C、PROLOG、Ada、C++、Java和Python等。用较早期的高级语言——FORTRAN77语言编写的程序清单如下：

```
*REVISED PROGRAM TO COMPUTE THE UNIT COST
*AND THE UNIT PRICE FOR A GIVEN PROFIT MARGIN.
*(IN THIS VERSION THE EPRESSIONS ARE WRITTEN AS PART OF THE PRINT STATEMENTS.)
*
REAL UNITS, TCOST, PMARG, UCOST, PRICE
```



```
*  
READ*, UNITS, TCOST, PMARG  
PRINT*, 'UNITS:', UNITS  
PRINT*, 'TOTAL COSTS:', TCOST  
PRINT*, 'PROFIT MARGIN:', PMARG, 'PERCENT'  
PRINT*, 'UNIT COST:', TCOST/UNITS  
PRINT*, 'UNIT PRICE:', (TCOST/UNITS) / (1-PMARG/100)  
END
```

可以看出,用高级语言编写的程序,程序员能够很容易理解,且容易学习,通用性强,书写出的程序比较短,便于推广和交流。不过,高级语言和计算机能够识别的机器语言已经相差甚远,要使其能够被计算机理解并执行,就需要将其转换为计算机能够识别的二进制数“0”和“1”,这就使其运算效率成为所有计算机语言中最低的。但是,因为计算机运行速度的飞速提升,高级语言运算效率低并没有成为其发展、壮大的瓶颈。现在,大多数计算机程序都是使用高级语言来设计的。

1.2.4 高级语言的执行方式

计算机是不能直接执行用高级语言编写的程序的。这就需要人们在定义好一门高级语言后,再开发出一套实现它的软件,这种软件被称作高级语言系统,其实质上是此高级语言的实现。高级语言的基本实现技术有编译和解释两种,下面对两者进行简单介绍。

1. 采用编译方式实现高级语言

编译方式是最常用的一种高级语言实现技术,它针对具体的高级语言(例如 C 语言)开发出一个翻译软件。这个翻译软件的功能是将采用此高级语言编写的程序翻译为所用计算机的机器语言的等价程序。把高级语言编写的程序翻译为机器语言等价程序的过程称为编译,类似于人们日常生活中的翻译。

对用这种高级语言编写好的程序,只需将其送给编译程序,就能得到与之对应的机器语言程序。此后,只要命令计算机执行这个机器语言程序,计算机就能完成人们所需要的工作,编译过程如图 1.4 所示。

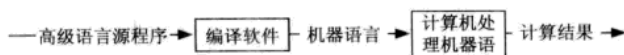


图1.4 高级语言编译过程

2. 采用解释方式实现高级语言

解释方式针对某种高级语言(例如 BASIC 语言)开发一个解释软件,此软件的功能就是读入这种高级语言编写的程序,解释所遇到的高级语言程序中的每个操作并且完成这些操作所要执行的动作。有了这种解释软件,只需直接将用高级语言编写好的程序送给运行此软件的计算机,就可以完成该程序所要的工作,解释过程如图 1.5 所示。



图1.5 高级语言解释过程



1.3 C 语言概述

C 语言是一门优秀的程序设计语言，诞生已有 30 多年的时间，因其具有诸多特点，现在仍然被广泛使用。本节从 C 语言的名称入手，简单地介绍 C 语言的发展历史、特点和应用范围。

1.3.1 为什么叫“C 语言”

C 语言的祖先就是“A 语言”，其全称为 ALGOL 60 语言。后来，在其基础上出现了一门称为 BCPL 的高级语言，也就是 B 语言。再后来，有人对 B 语言进行了改进，产生了一门新的高级语言，并取了 BCPL 语言的第二个字母作为新语言的名称，即 C 语言。

1.3.2 C 语言的版本

目前，最流行的 C 语言有 Microsoft C（简称 MSC）、Borland Turbo C（简称 Turbo C），以及 AT&T C。这些 C 语言版本不仅实现了 ANSI C 标准，而且在此基础上各自作了一些扩充，使其更加方便、完美。在 C 语言的基础上，1983 年又由贝尔实验室的 Bjarne Stroustrup 推出了 C++ 语言，C++ 语言进一步扩充和完善了 C 语言，成为一种应用广泛的面向对象的程序设计语言。

1.3.3 C 语言的特点

1. C 语言的优点

C 语言之所以这样“生机盎然”，到目前为止仍然被广泛使用，是因为其具有以下优点：

☑ 简单易学：

C 语言语法简单，程序书写自由，初学者非常容易接受，在较短的时间内就可以开始编程。

C 语言只提供了 32 个关键字（见附录 I）和 9 种控制语句，其主要用小写字母表示。学习和记忆这些关键字和语句非常容易且省时。与其他高级语言（如 Pascal 等）相比，用 C 语言编写的程序更加简洁、易懂。

☑ 结构科学：

C 语言是一门结构化语言。所谓结构化语言就是将自然语言加上程序设计语言的控制结构构成的语言，它专门用来描述加工逻辑。其显著的特点是将代码和数据分离，即程序的各个部分除了必要的信息交流外彼此独立。它能够把执行某个特殊任务的指令和数据从程序的其余部分分离出去、隐藏起来，可使程序层次清晰，便于使用、维护和调试。

C 语言是以函数形式提供给用户的，这些函数具有多种分支、循环语句控制程序流向，方便易用的函数定义和使用机制使人们可以把复杂的程序分解成若干个具有一定独立性的函数，以分解程序的复杂性，使程序设计工作更加简单、高效。

C 语言还提供了一套预处理命令，支持程序或软件系统的分块开发。利用这些机制，可以将一套复杂的软件系统分割成若干个简单的部分，每个部分由几个人或几个小组分别开发，然后再集成，构成最终系统。这种工作方式大大降低了软件开发工作的难度并提高其效率，使人们能够使用 C 语言开发出许多大规模的系统。



☑ 内容丰富:

C 语言首先提供了丰富的标准函数库。标准函数库是在总结其他语言基础上提出的,它将程序设计中许多需要的功能放在其中实现,比如输入/输出的功能。程序员在编程时,如果用到相关功能,只需调用标准函数库中相应的函数即可。这样就极大地提高了程序设计工作的效率,并且降低了其出错率。

C 语言还提供了丰富的运算符,其中共有 44 种运算符(见附录 I)。C 语言把括号、赋值、逗号、强制类型转换等都作为运算符处理,从而使其运算类型极为丰富,同时由运算符构成的表达式类型也多种多样。使用它,可以实现许多复杂的、其他高级语言难以实现的运算。

C 语言也提供了丰富的数据类型。包括整型、实型、字符型、布尔型、数组类型、指针类型、结构体类型、共用体类型等,可以实现各种诸如链表、树、栈、堆等复杂的数据结构。特别是引入了指针的概念,使程序效率大大提高。

☑ 运行效率高:

C 语言允许直接访问物理地址,可以直接对硬件进行操作。可以说,C 语言既具有高级语言的功能,又具有低级语言的许多功能,能够像汇编语言一样对位、字节和地址等计算机最基本的工作单元进行操作。这正是其运行效率高的原因所在。


人们之所以到现在为止,仍然在一些地方使用汇编语言,就是因为用高级语言写出的程序效率低。所以,对于开发效率要求特别高的程序,在程序设计时就只好使用汇编语言。C 语言程序生成代码质量高,程序执行效率高。它提供了一组比较接近硬件的低级操作,可用于写较低级、需要直接与硬件打交道的程序,所以常被用作汇编语言的“替代物”,从而不仅大大提高了开发低层程序的效率,又使得程序不是很难懂。用 C 语言编写的程序一般只比汇编程序生成的目标程序效率低 10%~20%。

☑ 可移植性强:

所谓可移植性,并不是指所开发的程序不做任何修改就可以在任何计算机上运行,而是指当条件有变化时,程序无须做很多修改就可运行。

汇编语言虽然运行速度快,但是移植性差。与汇编语言相比,C 语言有一个突出的优点就是适合于多种操作系统,如 Windows、DOS、UNIX,也适用于多种机型,如微机和工作站。C 语言是有力的和便于移植的,例如,大部分实用、便于移植的 UNIX 操作系统都是用 C 语言编写的。

在 UNIX 系统中,C 语言之外的其他语言,如 FORTRAN、APL、Pascal、LISP 和 BASIC 的编译或解释程序也是用 C 语言编写的。所以,在安装了 UNIX 操作系统的计算机上使用 Pascal 时,最终用 C 程序产生最后的可执行代码。

 **注意:** 此处所说的 C 语言的可移植性好,主要是将其和汇编语言相比较,与其他更高级的语言(如 Java 语言)相比,其可移植性并不出众。

2. C 语言的缺点

当然,C 语言并非“尽善尽美”,也存在以下不足:



- ☑ 与许多面向对象的语言（如 C++）相比，C 语言并不是一门数据封装性很好的语言，从而使其在保护数据的安全性上有很大缺陷。
- ☑ C 语言的语法限制不太严格，对变量的类型约束不严格，从而使程序的安全性受到很大影响。比如，对数组下标越界不进行检查，经常会引起错误。从应用的角度来看，C 语言比一些其他高级语言要难掌握。

1.3.4 C 语言的应用

因为 C 语言具有诸多优良特性，所以它几乎可以应用到所有的程序设计工作中，主要应用在以下 3 个方面。

1. 开发系统软件

此处所谓的系统软件，指计算机操作系统和系统使用的程序。因为 C 语言设计之初就被用来开发 UNIX 操作系统，所以其在开发操作系统软件方面具有得天独厚的优势。许多常用的操作系统（如 Windows、Linux 等）内核程序都是使用 C 语言编写的。

所谓的内核是指操作系统最基本的部分，是为众多应用程序提供对计算机硬件安全访问的一部分软件，这种访问是有限的，并且内核决定一个程序在什么时候对某部分硬件操作多长时间。而且，因为 C 语言可以很方便地和硬件进行交互，所以，许多操作系统使用的程序都是用 C 语言开发的。

2. 开发嵌入式软件

所谓嵌入式软件，是指用于执行独立功能的专用计算机软件。它一般不依赖操作系统，而直接和硬件交互，这就要求程序的运行效率很高，因此一般都是使用汇编语言开发的。但是因为汇编语言与高级语言相比易读性差，而 C 语言既能与硬件直接交互，又有很好的易读性，所以很多嵌入式软件也用 C 语言开发。

3. 开发应用软件

C 语言不仅具有绘图能力强、可移植性强等特点，还具备很强的数据处理能力，因此不仅适合编写系统软件，也适合编写诸如二维、三维图形和动画、数值计算、教学等方面的应用软件。

1.4 第一个 C 程序

经过前面的介绍，你是否已经“蠢蠢欲动”，想要编写一个 C 程序（用 C 语言开发的程序）？本节就从一个非常简单的 C 程序“Hello, World”入手，介绍 C 程序以及 C 程序的结构特点和运行过程。

1.4.1 为什么选“Hello, World”

许多介绍程序设计语言的书籍都以一个称为“Hello, World”的程序作为第一个程序来介绍，在笔者看来，主要是出于以下原因：

- ☑ 简单：这是一个最简单的程序，即使没有任何编程经验的人，稍微琢磨一下就能看懂，非常容易理解。



- ☑ 全面：其虽然简单，却比较全面地涵盖了一门程序设计语言的许多组成部分，结构性好。
- ☑ 成就感：“Hello, World”的意思是“您好，世界”。可以形象地理解为：当编写完第一个程序后，对着外界发出“您好，世界”的声音。虽然在编程的道路上仍然“任重而道远”，但也使你在编程上有了一点小小的成就感。

1.4.2 “Hello, World”程序

这个叫做“Hello, World”的程序完成在计算机显示器屏幕上显示“Hello, World”字样的功能，其程序清单如下：

```
001      #include "stdio.h"
002      int main()
003      {
004          /*使用 printf()函数输出“Hello, World”*/
005          printf("Hello,World!\n");
006          return 0;
007      }
```

1.4.3 “Hello, World”程序解析

“Hello, World”程序可分为以下两个部分。

1. 预处理部分

程序的第 1 行属于预处理部分，其说明程序用到 C 语言系统提供的标准功能，为此要参考标准库文件“stdio.h”。也就是说，本行以下程序中要使用这个标准库里的文件。

2. 程序基本部分

第 1 行以后都是程序的基本部分，用来描述程序完成的具体工作。

第 2 行中的 main()表示“主函数”，在程序中有且只有一个。其前面的 int 表示这个函数的返回值类型。用花括号{}括起来的部分称为函数体，是函数要执行的内容。

第 4 行/*.....*/表示注释，其作用是告诉阅读程序的人下面一行的作用是输出“Hello, World”，它可以出现在程序的任何地方。

第 5 行是一个语句，其后要加一个分号，有关语句的细节将在第 3 章介绍。其中，printf 是 C 语言中的格式输出函数名，包含在前面引用的“stdio”标准库中。双引号内的字符串将原样输出，“\n”表示在屏幕上换行。而“return 0”表示主函数给系统一个返回值 0，表示其正常结束。

1.4.4 C 程序结构特点解析

通过前面几节的介绍发现，可以将一个 C 程序划分为几个不同的部分，每个部分的作用都不相同，也就是说，其结构非常清晰、易懂。具体来说，一个 C 程序具有以下结构特点：

1. C 程序由函数组成

C 程序其实是一种模块化的程序，可以看做是由若干个模块——函数组成的。可以将一个复杂的功能分解成若干个简单的功能，每个函数完成一种简单的功能，然后通过函数之间



的互相调用来实现最终的功能。

2. 一个函数由函数首部和函数体组成

☐ 函数首部，即函数的第一行。

函数首部包括函数的返回值类型、函数名、函数参数类型、函数形式参数。例如，下面是函数 `sum()` 的首部：

<code>int</code>	<code>sum</code>	<code>(int</code>	<code>x,</code>	<code>int</code>	<code>y)</code>
↓	↓	↓	↓	↓	↓
返回值类型	函数名	函数参数类型	形式参数名	函数参数类型	形式参数名

☐ 函数体，就是函数首部下面用 `{}` 括起来的部分。

如果一个函数内有多个 `{}`，则函数体是最外层的 `{}` 包含的内容。有关函数的细节将在第 5 章详细介绍。

3. 一个 C 程序有且只有一个主函数，而且总是从其开始执行

主函数就是名称为 `main()` 的函数，它在程序中有且只有一个，不管放在程序的什么地方，程序总是从它开始执行。具体地说，程序会从 `main()` 函数的函数首部下面的第一个 `{` 开始依次执行其函数体，其中可能会调用其他函数，直到执行到与前面对应的 `}` 结束。

4. 每个语句和数据定义的最后必须有一个分号

程序包括数据描述和数据操作。数据描述主要定义数据结构（用数据类型表示）和数据初值，由数据定义来实现。比如 `int a=1;`，就定义了一个 `int` 型的数据 `a`，其初值为 1。数据操作对已提供的数据进行加工，是由语句来实现的。比如 `a+1;` 就是一条用来对数据 `a` 做加一操作的语句。在 C 程序中，不管是数据定义还是语句，都必须以一个分号结束。

5. 程序书写格式自由

C 程序的书写格式非常自由，一行内可以书写多条语句或进行多个数据定义，一条语句也可以书写在多行上，而且没有行号。

6. 可以通过预处理调用标准库中的函数

C 语言提供了非常强大、完善的标准库。当编写程序时要用到标准库函数能实现的相关功能时，最好通过预处理操作来调用这些库函数，提高编程效率，同时也降低了程序错误率。其实，预处理还可以做其他的事情，有关预处理的细节将在第 8 章详细介绍。

7. 程序需要良好的注释

可以使用 `/*.....*/` 对 C 程序中的任何部分作注释。这种注释称为多行注释，也就是说此种注释可以跨行书写。它既不属于数据定义，也不属于语句。良好的程序必须加上必要的注释，以提高程序的可读性和易懂性。当程序修改后，也要保证注释的正确性、一致性和完整性，否则会给程序维护和完善工作留下致命的恶果。



注意：本书中多数注释采用 `/**` 的形式，这也是一种 C 程序的注释方式，称为单行注释，其只能注释一行语句或者数据定义，不能跨行。它不是标准 C 程序的注释方式，如果所使用的编译器不支持这种注释方式，请使用 `/*.....*/` 的注释方式。




1.4.5 C 程序是如何执行的

计算机只能识别和执行特定二进制形式的机器语言程序。而 C 语言是一门高级语言，用其编写的程序虽然容易使用、编写和阅读，但是不能被计算机识别和执行，必须选择使用编译或者解释的方式来转换成机器语言才能够被计算机执行。

1. 编译和连接

C 程序的转换过程包括编译和连接两个步骤，其执行过程如图 1.6 所示。

 **提示：**使用 C 语言编写的程序称为 C 语言源程序，在 Windows 操作系统中，它是扩展名为 .c 或者 .cpp 形式的文件。

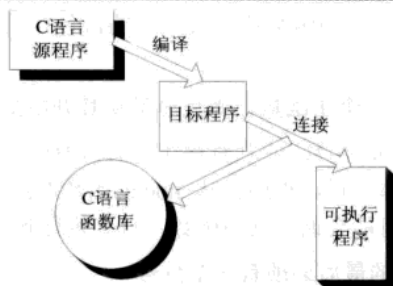




图1.6 C程序执行过程

第 1 步 编译。由编译程序对源程序文件进行分析和处理，生成相应的机器语言目标模块。由目标模块构成的代码文件称为目标文件。

 **提示：**在 Windows 操作系统中，目标文件扩展名为 .obj 形式的文件。

目标文件还不能执行，因为其中缺少 C 程序运行所需要的一个公共部分，即 C 程序的运行系统。此外，一般 C 程序里都要使用函数库提供的某些功能。例如，“Hello, World”程序中用到标准函数库的格式输出函数 printf() 函数。

第 2 步 连接。为构造出完整的可运行的程序，还需要第二步加工——连接。这一工作由连接程序完成，将编译得到的目标模块与其他必要部分（运行系统、函数库提供的功能模块等）拼装起来，产生可执行程序。

 **提示：**在 Windows 操作系统中，可执行程序是扩展名为 .exe 的文件。

对 C 源程序经过这两步处理后，就能得到一个与其对应的、可以在计算机上执行的程序。启动运行这个可执行程序，将能看到其执行结果。例如，“Hello, World”程序执行后将得到一行输出，通常显示在计算机屏幕上或者图形用户界面上的特定窗口里，如下所示：

```
Hello, World!
```

2. 什么时候使用集成开发环境

当编写好一个 C 语言源程序，需要将其转化为可执行程序时，可以直接用操作系统命令形式启动各种基本加工工作（启动编译系统、连接系统等），做法是先用一个命令要求编译源程序，再用另一个命令进行连接。其中除了要把源程序文件名作为命令参数外，还常常需要给出一些其他参数。

这些命令的书写形式比较复杂,使用不方便。此外,为了输入、编辑和修改程序,还需要另用一个编辑系统,这个过程相当复杂。为了提高效率,人们开发出了许多程序作为实现上述的输入、编辑、修改、编译、连接等功能的软件,其中就包括集成开发环境——Microsoft Visual C++ 6.0。

1.5 Microsoft Visual C++ 6.0 集成开发环境简介

“工欲善其事,必先利其器”。开发高质量的C程序也一样,首先要选择一种好的开发工具。在众多的开发工具中,Microsoft Visual C++ 6.0 凭借其优良特性,成为编写C程序及C++程序的首选工具。本节就对其进行简要的介绍。

1.5.1 集成开发环境 (IDE)

较早期程序设计的各个阶段都要使用不同的软件来进行处理:

- ☒ 使用字处理软件(如 Windows 的记事本)编辑源程序。
- ☒ 使用编译程序对源程序进行编译。
- ☒ 使用连接程序进行函数、模块连接。

依靠上述方式,开发者在编译、执行和调试程序时,必须在几种软件间来回切换操作。此方式对一些小型的软件开发来说还可以应付,但是对于大型软件来说,不仅效率低,而且出错率比较高。为了解决上述问题,于是产生了开发一种可以将编程过程中需要的所有软件集成在一起统一管理和使用的软件——集成开发环境。

集成开发环境(Integrated Develop Environment, IDE),是用于提供程序开发环境的应用程序,一般包括代码编辑器、编译器、调试器和图形用户界面工具。IDE 集成了代码编写、分析、编译和调试等开发软件所需的整套功能。所有具备这一特性的软件或者软件套(组)件都可以称为集成开发环境。如微软的 Visual Studio 系列, Borland 的 C++ Builder、Delphi 系列等。

集成开发环境可以独立运行,也可以和其他程序并用。例如, BASIC 语言可以在微软办公软件中使用,可以在微软 Word 文档中编写 WordBasic 程序。IDE 为用户使用 C、C++、Java 和 PowerBuilder 等现代编程语言提供了方便。

1.5.2 集成开发环境的功能

IDE 有 3 项必须集成的基本功能:

- ☒ 编辑器。
- ☒ 编译/连接器。
- ☒ 调试器。

除此之外,现代集成开发环境还包括有一些更强大的功能,如代码提示、项目管理、界面设计、建模功能等。

不同的技术体系有不同的 IDE。比如 Visual Studio 可以称为 C++、C、VB、C#等语言的集成开发环境,所以 Visual Studio 可以叫做 IDE。同样, Borland 的 JBuilder 也是一个 IDE,



它是 Java 语言的 IDE。Eclipse、GCC 等工具都具备基本的编码、调试功能，所以都可以称为 IDE。

IDE 系统一般采用窗口菜单技术，提供了专供编程用的编辑环境，通过菜单方式提供编译、连接以及启动可执行程序命令。利用 IDE 写程序，开发过程中的各种工作都很方便，能大大提高编程工作的效率。

许多 IDE 在不断改进和发展，逐渐成为一类复杂的软件系统。使用好这种系统也需要学习，只有掌握了 IDE 的功能，才能更好地发挥其作用。IDE 虽然能使编程工作更方便，但其没有也不可能改变这个工作的实质。编好程序的最基本也是最重要的因素仍然是程序员。这就好比做一道好菜虽然需要一把好菜刀，但是最终的菜好不好吃仍然取决于厨师的水平。



注意：即使在编程时使用了 IDE，程序的执行方式仍然没有变，比如一个 C 源程序，其执行过程仍然是编译生成目标文件，然后连接生成可执行文件，只不过这些工作都交给 IDE 去做而已。

1.5.3 为什么选择 Microsoft Visual C++ 6.0

1. C 程序开发工具简介

能够用来开发 C 程序的工具有很多，其中比较常用的有以下几种：

- ☑ GCC：一款用于在 UNIX 或 Linux 操作系统上开发 C 程序的工具。
- ☑ Turbo C（简称 TC）：一款由 Borland 公司开发的用于 MS-DOS 和 Windows 操作系统上的小型 C 程序开发工具。
- ☑ Microsoft Visual C++ 6.0（简称 Visual C++ 6.0）：一款由微软公司开发的用于 MS-DOS 和 Windows 操作系统上的 C/C++ 程序的集成开发工具。

这些工具集成了开发 C 程序时的输入、编辑、修改、编译、连接等功能，从而很好地帮助程序员开发满足需要的程序。本书中选用 Microsoft Visual C++ 6.0 作为所有 C 程序的开发工具，一方面是因为本书中的程序都是在 Windows 系统下编写和运行的，另一方面是因为下面要介绍的 Visual C++ 6.0 的诸多特点。

2. Microsoft Visual C++ 6.0 的特点

- ☑ Visual C++ 6.0 是由 Microsoft（微软）公司开发的，可以使用功能强大的微软基础类库 MFC（Microsoft Foundation Classes），其充分体现了 Microsoft 公司的技术精华。并且因为 Microsoft 公司在操作系统市场上的垄断地位，使得使用 Visual C++ 6.0 开发出来的软件稳定性好、可移植性强，并且软件和硬件相互独立。
- ☑ Visual C++ 6.0 作为 Visual Studio 可视化组件家族中最重要的一个成员，与其他可视化工具如 Visual Basic 6.0、Visual J++ 6.0 以及 Visual C# 6.0 紧密地集成在一起，可进行不同类型程序开发工作，适合于特殊、复杂和综合软件项目的开发以及系统软件的开发。
- ☑ Visual C++ 6.0 源代码编辑器提供了自动语句完成功能，编辑输入源程序时，能自动显示当前对象的成员变量和成员函数，并指明函数的参数类型。

- ☑ Visual C++ 6.0 的编译器增加了新的编译参数,改进了对 ANSI C++ 标准的支持,并采用 Microsoft 的代码优化技术,使生成的目标代码更加短小,应用程序运行速度更快。
- ☑ Visual C++ 6.0 程序调试器功能更加强大,提供了诊断映射机制、无须重编译的调试、远程调试和实时调试等功能。
- ☑ Visual C++ 6.0 的联机帮助 MSDN Library (Microsoft Developer Network Library) 采用当今最流行的 HTML 格式。既能与集成开发环境有机地结合在一起,使得用户在编程时随时查询需要的内容,又能脱离集成开发环境而独立运行。用户还可以通过网络获取实时的帮助信息和实例。
- ☑ Visual C++ 6.0 通过 Visual Studio 为用户提供了一些其他的实用工具,如 Spy++ 查看器、ActiveX Control Test Container 控件测试容器及 Register Control 控件注册程序,极大地扩展了 Visual C++ 6.0 的功能。

1.5.4 Microsoft Visual C++ 6.0 的版本

Visual C++ 6.0 是 Microsoft 于 1997 年 4 月推出的 Visual C++ 编译器,其包括 3 个版本。各个版本之间的区别如表 1.1 所示。

表 1.1 Visual C++ 6.0 的不同版本

版 本	特 点
学习版 (Learning Edition)	除了代码优化、剖析程序(一种分析程序的运行时行为的开发工具)和导入 MFC 库的静态链接外,Visual C++ 6.0 学习版提供了专业版的其他所有功能。学习版的价格要比专业版本低很多,这是为了使学习 Visual C++ 6.0 的个人也可以负担得起。但这些人不可以使用 Visual C++ 6.0 学习版开发软件,其授权协议明确禁止这种做法
专业版 (Profession Edition)	Visual C++ 6.0 专业版可用来开发 Win32 应用程序、服务和控件。在这些应用程序、服务和控件中可使用由操作系统提供的图形用户界面或控制台 API
企业版 (Enterprise Edition)	可用来开发和调试为 Internet 或企业内网 (Intranet) 设计的客户端/服务器应用程序。在 Visual C++ 6.0 企业版中还包括开发和调试 SQL 数据库应用程序和简化小组开发的开发工具

1.5.5 Microsoft Visual C++ 6.0 的安装

在使用 Visual C++ 6.0 之前,必须对其进行安装。在 Windows XP 中安装 Visual C++ 6.0 简体中文企业版 (Enterprise Edition) 的过程如下:

第 1 步 在 CD-ROM/DVD-ROM 驱动器中插入 Visual C++ 6.0 系统光盘。



图 1.7 启动文件图标

第 2 步 双击 SETUP.EXE 文件,此文件图标如图 1.7 所示。

第 3 步 打开 Visual Studio 6.0 企业版安装向导,如图 1.8 所示,单击 Next (下一步) 按钮。若单击 View Readme (浏览帮助文件) 按钮则会弹出窗口显示帮助文档。



第4步 弹出 End User License Agreement (最终用户许可协议) 对话框, 如图 1.9 所示。选择 I accept the agreement (接受协议) 单选按钮, 然后单击 Next (下一步) 按钮。

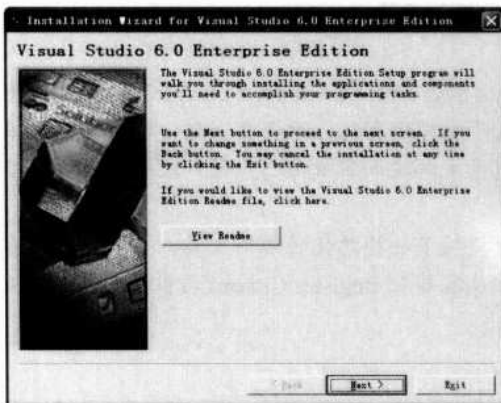


图1.8 安装向导

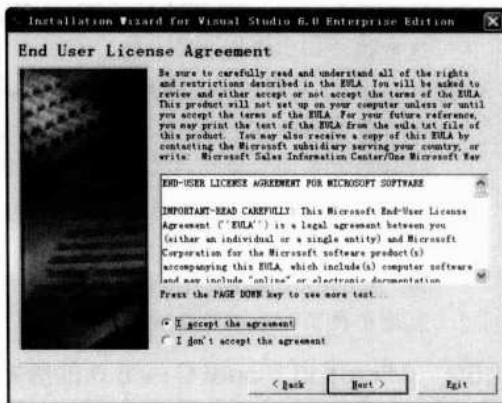


图1.9 End User License Agreement对话框

第5步 弹出 Product Number and User ID (产品号 and 用户 ID) 对话框, 如图 1.10 所示。输入产品 ID 号、姓名和公司名称, 然后单击 Next (下一步) 按钮。

第6步 弹出 Visual Studio 6.0 Enterprise Edition (Visual Studio 6.0 企业版) 对话框, 如图 1.11 所示。选择 Custom (自定义) 单选按钮, 然后单击 Next (下一步) 按钮。

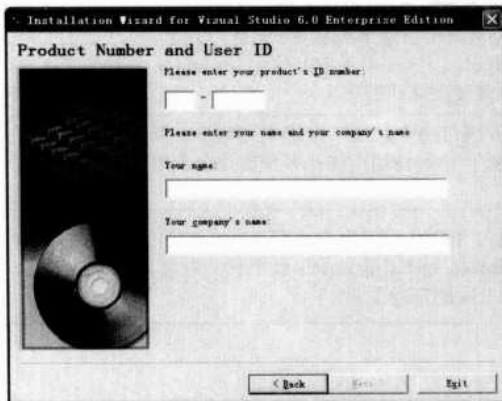


图1.10 Product Number and User ID对话框

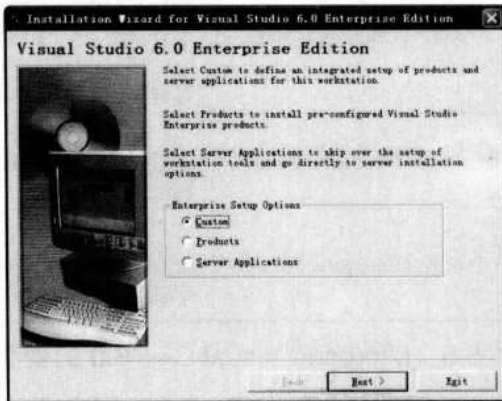


图1.11 Visual Studio 6.0 Enterprise Edition对话框

第7步 弹出 Choose Common Install Folder (选择共用安装文件夹) 对话框, 如图 1.12 所示。用户可以选择自己想要安装的文件夹, 然后单击 Next (下一步) 按钮。

第8步 等待片刻后, 弹出 Visual Studio 6.0 Enterprise Setup (Visual Studio 6.0 企业版安装程序) 对话框, 如图 1.13 所示。单击 Continue (继续) 按钮。

第9步 在弹出的对话框中, 单击 OK (确定) 按钮, 如图 1.14 所示。

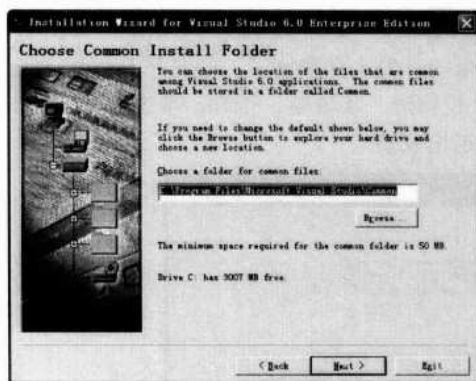


图1.12 Choose Common Install Folder对话框

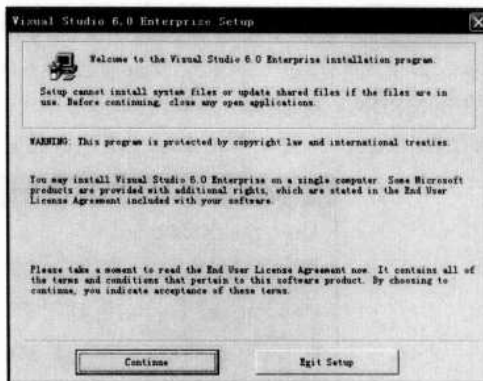


图1.13 Visual Studio 6.0 Enterprise Setup对话框

第10步 等待片刻，弹出 Visual Studio 6.0 Enterprise-Custom（Visual Studio 6.0 企业版一用户）对话框，如图 1.15 所示。在 Options（选项）列表框中选择 Microsoft Visual C++6.0 和 Data Access 复选框后，单击 Continue（继续）按钮。由此便开始安装，屏幕上出现安装进度指示对话框，如图 1.16 所示。

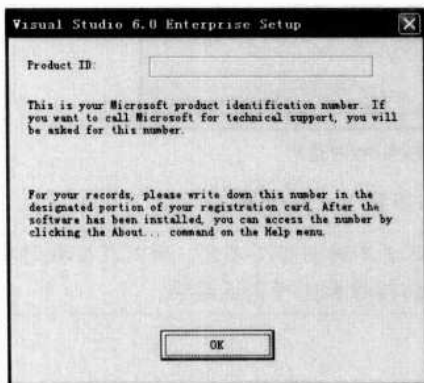


图1.14 单击OK按钮

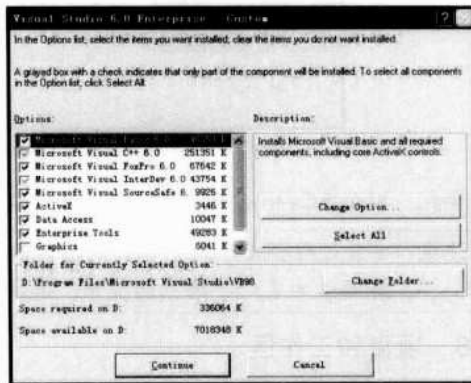


图1.15 Visual Studio 6.0 Enterprise-Custom对话框

第11步 安装完组件后，会弹出对话框要求重启 Windows，如图 1.17 所示。单击 Restart Windows（重新启动 Windows）按钮，计算机会自动重启。

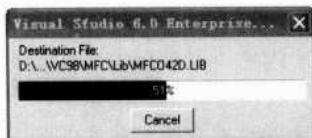


图1.16 安装进度提示对话框

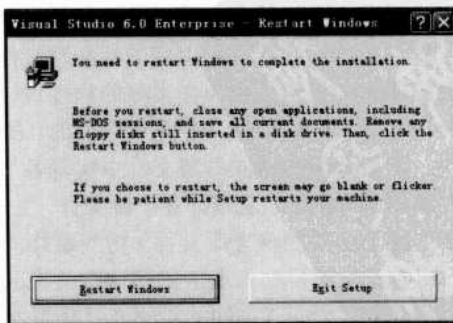


图1.17 重启Windows对话框



第 12 步 重新启动计算机后，便进入 Install MSDN（安装 MSDN）对话框，如图 1.18 所示。如果用户想进一步安装 Visual C++ 6.0 所带的开发文档和例子，则选择 Install MSDN（安装 MSDN）复选框，并单击 Next（下一步）按钮进一步安装。否则，单击 Exit（退出）按钮退出安装。此处，单击 Exit（退出）按钮，退出安装。

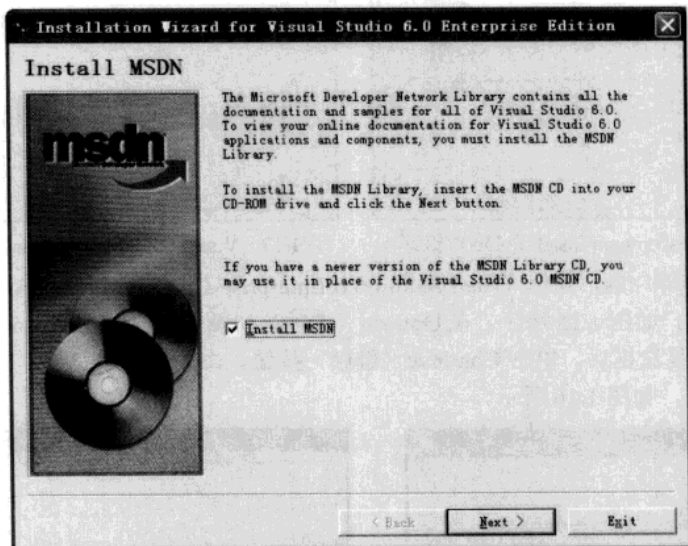



图1.18 Install MSDN 安装MSDN对话框

至此，Visual C++ 6.0 安装完成，整个过程大约需要 5~30 分钟。

 **注意：**安装过程会由于用户安装的版本和软件环境不同而有所不同，所以在安装时要认真阅读安装向导给出的提示，选择正确的按钮和选项完成安装。

1.5.6 项目和工作区

在使用 Visual C++ 6.0 时，有两个非常重要的概念，即项目和工作区。本节就对其进行介绍。

1. 项目简介

用来生成一个最终程序的各个源程序文件和其他辅助文件的集合被称为一个“项目”。例如，将一个源程序和其要引用的标准库文件放在一起组成一个集合就叫做一个项目。

提出项目的概念是因为在实际的程序设计工作中，一个程序的结构往往十分复杂，如果想用一个文件来实现的话几乎是不可能的。所以，程序员往往把一个程序分成若干较小的功能模块，然后分别在不同的源文件中实现各个模块的功能。

在编译时，首先分别编译各个源文件，生成一系列的目标文件，然后再将其相互连接（往往还要连接上必要的静态库文件）得到最终的程序文件。为了维护一个项目中文件间的相互关系，其中一般还要加入用于描述这些关系的项目描述文件，其内容和格式根据编译器的不同也不尽相同。对于 Visual C++ 6.0 是项目文件夹中后缀名为 .dsp 的文件。此文件的内容是由 Visual C++ 6.0 自动维护的，不需要也不应当被程序员修改。

2. 项目的实例

下面就列举一个项目的实例来介绍一下 Visual C++ 6.0 中项目的组织结构。例如，“FirstC”项目的文件夹结构如图 1.19 所示。

“FirstC”项目只是一个简单的项目，在一般情况下，Visual C++ 6.0 的项目组成是十分复杂的。但是需要说明的是，Visual C++ 6.0 对项目有良好的自动维护机制，因此很多文件不需要用户亲自来维护。甚至在一个项目中，用户一般都很少自己去新建或打开文件，而是提出具体要求让 Visual C++ 6.0 来判断是否应该新建文件并负责建立文件。

例如，当用户要新建一个类（C++ 中的概念，后面会介绍到）的时候，Visual C++ 6.0 会自动为这个类建立一组源文件（.h 与 .cpp 文件）。同时，当用户要查看某一段程序时，也不需要一个个地打开文件然后查找，只需要在类视图窗格中选中想要阅读的函数，Visual C++ 6.0 则会自动帮用户找到其所在的文件并打开。可以形象地认为，Visual C++ 6.0 在这里就承担了一个大管家的角色，帮助用户管理了复杂的项目文件结构，把逻辑层面上的内容清晰地呈现在用户的面前。

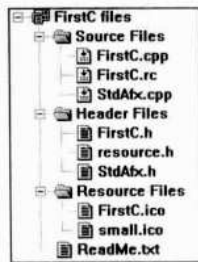


图1.19 FirstC项目文件夹结构图

3. 工作区

工作区的概念可以认为是 Visual C++ 6.0 项目管理机制对项目概念的一个延伸。在实际编程工作中，很多时候用户不仅仅只关心各个文件的内容。例如，你刚刚在某个文件中增加了一个函数，下班时间就到了，但是这个函数还没有写完，你希望再次开始工作的时候能自动打开此文件并把输入位置设置为上次中断的位置而不是文件开头以便于继续工作。此时，可利用 Visual C++ 6.0 的“工作区”来满足此需求。

“工作区”的概念正如其名，用于描述用户当前工作状态。例如，打开了哪些文件、当前输入位置，此类信息被保存在工作区文件中，当 Visual C++ 6.0 关闭的时候会自动保存。当用户下一次继续工作的时候，只需要打开这个工作区文件，就可以回到上一次关闭前的状态，而不需要再费心思回忆上一次到底编写或者修改到哪一行代码。此处要特别注意以下两点：

- 工作区与项目并不一定是一一对应的。一个工作区中也可以包含一个以上的项目。但一般情况下，用户还是喜欢每次只在一个项目中进行工作，因此一个工作区中往往只有一个项目。
- 正因为如此，当用户想打开一个项目的时候，常用“打开工作区（Open Workspace）”菜单项打开这个项目文件夹中的工作区文件而不是使用“打开项目（Open Project）”菜单项。

1.5.7 Visual C++ 6.0 界面简介

单击 Windows 的“开始”按钮，从“开始”菜单启动 Visual C++ 6.0 进入 Developer Studio 开发环境。打开一个项目后，Developer Studio 由标题栏、菜单栏、工具栏、视图窗格、源代码编辑窗格、输出窗格和状态栏组成，Visual C++ 6.0 界面如图 1.20 所示。

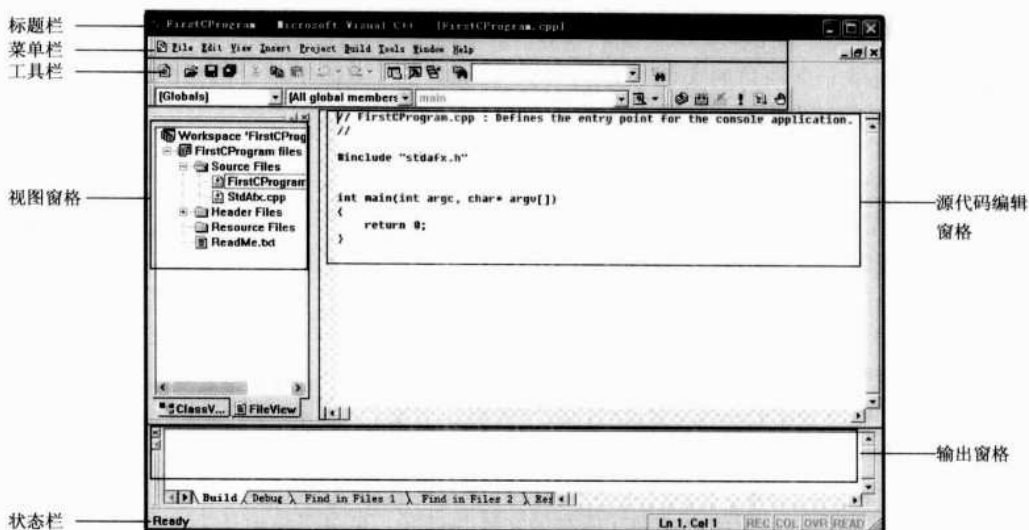


图1.20 Visual C++ 6.0界面

☑ 屏幕最上端是标题栏:

标题栏用于显示应用程序名和所打开的文件名,标题栏的颜色用于表明对应窗口是否为激活的。

☑ 标题栏左端为控制菜单框:

控制菜单是用于打开窗口控制菜单的图标。用鼠标单击该图标,将弹出窗口控制菜单。窗口控制菜单用于控制窗口的大小和位置,如还原、移动、关闭、最大化和最小化等。标题栏的右边有3个控制按钮,从左至右分别为最小化按钮、还原按钮和关闭按钮,这些按钮用于快速设置窗口大小。例如,使窗口填充整个屏幕、将窗口最小化为图标或关闭窗口。

☑ 标题栏的下面是菜单栏和工具栏:

菜单栏中的菜单项包括了 Visual C++ 6.0 的全部操作命令,工具栏以位图形式显示常用的操作命令。一些不常用的工具栏一般情况下不出现在主窗口中,只有使用时才会自动弹出。Developer Studio 中的菜单栏和工具栏都是停靠式的,可以用鼠标拖动到屏幕的任何位置,其大小也可人工进行调整。

☑ 工具栏的左下方是视图窗格:

视图窗格中包括类视图、资源视图和文件视图,后面的小节会对其进行详细介绍。

☑ 工具栏的右下方是源代码编辑窗格:

源代码编辑窗格用于显示当前编辑的 C/C++ 程序源文件和资源文件。编辑窗格包含“最大化”、“最小化”、“关闭”按钮和系统菜单的普通框架窗格。当打开一个源文件或资源文件时,就会自动打开其编辑窗格。可以同时打开多个编辑窗格,编辑窗格以平铺式或层叠式显示。

☑ 视图窗格和编辑窗格下面是输出窗格:

输出窗格用于显示项目建立过程中所产生的信息。当编译、连接时,会显示编译和连接信息。当进入程序调试状态时,主窗口中还会出现一些调试窗格。



☑ 屏幕底端是状态栏:


给出当前操作或所选择命令的提示信息、当前光标所在位置信息,以及当前的编辑状态信息等。

1.5.8 常用菜单项

可以看出, Visual C++ 6.0 的界面上有很多菜单项,不过,只有很少菜单被经常用到,因此本小节仅对常用的两个菜单进行详细介绍。

1. File (文件) 菜单组

- ☑ New (新建): 新建项目或新建文件,选择此菜单项后会出现一个对话框询问新建项目还是新建文件以及新建的项目或文件的类型。
- ☑ Open (打开): 用于打开文件。此功能一般不常用到,但有的时候希望打开一些不属于任何项目的源文件时可以用到此项功能。

 注意: 此功能只是用来打开单独的文件,其默认打开类型是.h 和.cpp 文件。

- ☑ Close (关闭): 关闭当前被打开且处于活动状态的文件。活动状态就是目前正在操作的状态。
- ☑ Open Workspace (打开工作区): 在打开项目时常用此菜单项,用其来打开项目文件夹中的.dsw 文件。也可以用来打开项目描述文件(.dsp),但是如果检测到有.dsw 文件的存在则会自动以打开这个.dsw 文件的方式打开项目。
- ☑ Save Workspace (保存工作区): 保存当前的工作区状态。此功能不常用,因为 Visual C++ 6.0 在关闭的时候往往会自动执行这个操作。
- ☑ Close Workspace (关闭工作区): 当用户需要开始或打开一个新项目时,最好先关闭当前的工作区,此时应该使用此菜单项。如果仅仅关闭所有打开的文件是不行的,因为工作区仍然处于打开状态。
- ☑ Save (保存): 保存当前处于被编辑状态的源文件。此功能作用并不是很大,因为它一次只能保存一个文件,而且只能保存在工作区中以文本方式打开(.h 和.cpp 等)的文件,如果对资源描述一类的文件进行了修改(当然,是通过资源编辑器,而不是直接用文本方式修改)那就无能为力了。
- ☑ Save As (另存为): 存在与 Save (保存)命令一样的缺点。
- ☑ Save All (保存全部): 保存目前打开的所有文件,包括以非文本方式打开的文件(如.rc、.aps 等),即保存目前对该项目所做的一切修改。此命令经常被使用。

其他的菜单项和通用的菜单项没有太大的区别,在此不再详述。

File (文件) 菜单项有以下两点需要特别注意:

- ☑ 其中的 Recent Files (最近的文件) 和 Recent Workspace (最近的工作区) 是两个选项,一个对应于打开文件,一个对应于打开工作区,后者比前者常用得多。
- ☑ 如果当前被激活的文件中存在未被保存的修改,则在标题栏中会出现一个“*”提示符。



2. Compile (编译) 菜单项

- ☑ Compile XXX (编译当前文件): 编译当前的源文件产生与其对应的.obj 文件。
- ☑ Build XXX (构建项目的 EXE 文件): 首先编译所有没编译过或已被修改过的源文件, 然后连接其.obj 文件和其他的文件生成最终的 EXE 文件。常用它来实现构建。
- ☑ Rebuild All (重建全部): 同样用来构建 EXE 文件。和构建菜单的区别是: 构建菜单只编译过时的.obj 文件, 而不考虑目前这些.obj 文件是否为最新版本, 重新编译所有源文件后连接生产工具生成 EXE 文件。一般很少使用这个功能。
- ☑ Execute XXX (执行 EXE 文件): 首先构建出 EXE 文件, 然后运行。推荐使用。
- ☑ Set Active Configuration (设置可运行配置): Visual C++ 6.0 程序一般有两个编译配置, 即 Debug (调试版) 和 Release (完全版), 新建的项目默认为 Debug 版。此配置编译出的 EXE 文件在项目文件夹的 Debug 目录中, 一般比较大, 但是包括了很多可调试信息, 方便与 Visual C++ 6.0 调试器一起完成项目的调试。而 Release 版的程序则小得多, 但不能调试, 因此一般是作为项目的最终成品, 而在创建工程中一般使用 Debug 版。

1.5.9 常用工具栏

工具栏由一些操作按钮组成, 分别对应着某些菜单选项或命令的功能, 可以直接单击这些按钮来完成指定的功能。工具栏按钮大大简化了用户的操作过程, 并使操作过程可视化。

Visual C++ 6.0 包含有十几种工具栏。默认状态下, 屏幕工具栏区域显示两个工具栏, 即 Standard 工具栏和 Build 工具栏。本节只对这两个工具栏进行介绍。

1. Standard 工具栏

Standard 工具栏如图 1.21 所示。



图1.21 Standard工具栏

Standard 工具栏中的各个按钮的功能描述如表 1.2 所示。

表1.2 Standard工具栏中各按钮功能描述表

按钮名称	功能
New Text File	创建新的文本文件
Open	打开已有的文档
Save	保存文档
Save All	保存所有打开的文件
Cut	剪切选定的内容到剪贴板中
Copy	复制选定的内容到剪贴板中
Paste	在当前插入点处插入剪贴板中的内容
Undo	取消最后的操作

(续表)

按钮名称	功能
Redo	重复先前取消的操作
Workspace	显示或者隐藏工作区窗口
Output	显示或者隐藏输出窗口
Window List	管理当前打开的窗口
Find in Files	在多个文件中搜索字符串
Find	激活查找工具
Search	搜索联机文档

2. Build 工具栏

Build 工具栏如图 1.22 所示。



图1.22 Build工具栏

Build 工具栏中的各个按钮的功能描述如表 1.3 所示。

表1.3 Build工具栏中各按钮功能描述表

按钮名称	功能
Select Active Project	选择活动项目
Select Active Configuration	选择活动配置
Compile	编译文件
Build	建立项目
Stop Build	停止项目的建立
Execute Program	执行程序
Go	启动或者继续执行程序
Inserts Remove Breakpoint	插入或者删除断点

如果要在屏幕上显示或者隐藏工具栏,可在屏幕工具栏区域中右击,从弹出的快捷菜单中选择或者清除相应的工具栏。

1.5.10 视图窗格简介

当 Visual C++ 6.0 中有活动的项目时,其界面的左侧会出现一组选项卡,从左至右分别是 Class View (类视图)、Resource View (资源视图) 和 File View (文件视图)。

1. 类视图

Visual C++ 6.0 的类视图以树形图的形式显示了在当前项目中的类层次结构,如图 1.23 所示。

- ☑ 刚打开一个项目时,所有的类都以层叠的方式显示,即只显示类名,并在前面有一个“+”图标。



- ❑ 在所有的类下面有一个“Globals (全局)”选项，其中包括了不在任何类中的函数和全局变量。双击一个类名则会自动在代码编辑窗格中转到这个类定义的位置。若要进一步查看该类的内容，可以单击类名左侧的“+”图标。此时该类层次结构被展开，显示出它的所有成员函数和成员变量，同时前面的图标变为“-”，单击其图标则重新回到折叠状态。
- ❑ 函数前面以粉红色的小方块图标标识，而变量前面则是一个绿色的小方块标志。
- ❑ 在小方块的前面有一个锁的图标，则表示该成员是私有 (Private) 的，若为一个钥匙的图标，则表示该成员是被保护 (Protect) 的，若没有其他图标，则表示该成员是公共 (Public) 的。
- ❑ 双击一个函数的名字，则编辑窗格的输入位置跳转到该函数的定义 (实现) 位置，若想跳转到其声明位置，可以右击类视图中的函数名，在弹出的快捷菜单中选择 Go To Declaration (跳转说明) 命令；选择 Property (属性) 命令则可以直接查看该函数的属性，包括返回值类型、参数类型等。如果双击一个变量名，则会跳转到该变量的定义位置，同时也可以使用 Property (属性) 右键菜单来查看这个变量的类型。
- ❑ 若想新建类，则可以在类视图中的根节点 (显示为 XXX classes, XXX 为当前项目名) 的右键菜单中选择 New Class (新建类) 命令，之后在弹出的对话框中填写好相应的内容，单击 OK 按钮即可。之后 Visual C++ 6.0 会帮你产生相关的文件和类的最初定义信息，并将当前输入位置切换到新建的类实现文件中。
- ❑ 要在类中新建函数或变量，可以在类名的右键菜单中选择 Add Member Function (添加成员函数) 或 Add Member Variable (添加成员变量)，填写好相应内容后，Visual C++ 6.0 会自动产生代码并跳转输入位置。若要删除一个函数，特别是消息处理函数或映射函数，最好使用函数名右键菜单中的 Delete (删除) 命令，它不仅会清除掉函数的实现和声明，还会同时清除掉消息映射宏中的相关项。当然，除此之外，使用传统的纯手工方法也是可以的，只是前者工作量小而且不容易出错而已。

类视图中显示的内容是由 Visual C++ 6.0 自动维护的，会依用户的修改自动完成更新，并在退出 Visual C++ 6.0 时自动保存。

2. 文件视图

文件视图是 Visual C++ 6.0 界面左侧的第 3 个选项卡。它也是以树形的形式显示的，如图 1.24 所示。

文件视图中的文件虽然也是以类似 Windows 资源管理器中的文件夹树形显示的，但此处显示的文件层次关系并不是实际存放的文件树，而是项目中各文件间的逻辑关系。每一个项目都有 4 个逻辑文件夹：

- ❑ Source Files (源文件)。
- ❑ Header Files (头文件)。



图 1.23 类视图



图 1.24 文件视图

☑ Resource Files (资源文件)。

☑ External Dependencies (外部依赖)。

另外,还有一些不属于任何逻辑文件夹的文件,比如 ReadMe.txt 等,这些文件一般不需要开发者维护。源文件中包括了程序中所有以.cpp 和.c 为后缀名的文件。头文件是项目中的头文件(.h)。资源文件是项目中的资源文件,要编辑资源文件,一般不需要在这里打开,而可以在 Resource View (资源视图)选项卡中选择相应的 ID 号来对其进行修改。外部依赖的文件是项目的外部依赖文件。

例如,某个项目中有一句“include stdio.h”,而 stdio.h 并不是项目中的头文件,则这个文件被认为是外部依赖的。刚打开项目的时候,Visual C++ 6.0 一般不知道项目中有哪些外部依赖的文件,必须构建一次项目,这个文件夹中的内容才会被更新。有时候可能会发现在一些文件名的后面有一个“*”符号,这表示在这个文件中有未被保存的修改存在。

在文件视图中双击文件名则会打开这个文件,若其已经被打开,则将输入焦点切换到该文件。在文件名的右键菜单中选择 Delete 命令可以将这个文件移出项目,但必须注意一点,这个文件并没有被删除,也没有被移动,只是被认为不再属于该项目。如果要真正删除该文件,还需要在 Visual C++ 6.0 外面删除这个文件。这一点在需要重写一个文件的时候要特别注意,如果仅在文件视图中将其删除,可能会造成一些奇怪的错误。

3. 资源视图

资源视图是 Visual C++ 6.0 界面左侧 3 个选项卡中的一个,它也是以类似文件视图的逻辑文件层次树的形式显示的,如图 1.25 所示。

资源是 Windows 下编程的新概念。众所周知,Windows 程序是因其精致的 GUI (Graphic User Interface, 图形用户界面) 著称的。但是如果所有的图形都要在程序中通过绘图命令来实现,那工作量简直是不可想象的。除此之外,很多情况下,用户可能会希望程序能播放一些声音提示,如果将其以文件的形式存放在程序之外当然也未尝不可,但不如把它们直接写到程序里面方便。因此,Windows 下的程序引入了资源的概念,资源一般是一些图片、图标或其他程序中需要直接使用的非代码的组成部分,它们被存放在生成的 EXE 文件的“Resource Section (资源节)”中。

项目中的每一个资源平时都是单独以相应的文件形式存放在 Res 文件夹中的,而它们之间的关系则存放在.rc 文件中。在编译时被编译成一种资源目标文件,然后和.obj、.lib 文件等一起连接起来组成最终的 EXE 文件。

在项目中,每个资源用一个 32 位无符号整数常量标识,为了便于记忆,用#define 宏定义了相应的符号常量,比如 IDI_MAINFRAME 等,这被称为资源 ID (Resource Identifier)。这些宏定义存放在 resource.h 头文件中,所以一般可以在很多.cpp 文件中看到#include "resource.h" 的命令,这就是为了使其可以识别这些常量宏定义。resource.h 是由 Visual C++ 6.0 维护的,当用户新增、删除资源或更改资源 ID 时,它会被自动更新。至于使用资源的方式,Windows API 和 MFC 都提供了很多相应的函数。



图1.25 资源视图



1.5.11 代码颜色

在 Visual C++ 6.0 中,代码的颜色除了起装饰作用外还显示了许多信息。在 Visual C++ 6.0 中代码主要有 4 种颜色:黑、蓝、绿和灰。各种颜色显示的信息如下:

- ☒ 黑色是最常见的颜色,所有普通的代码是用黑色表示的。
- ☒ 蓝色则标识关键字,包括 if、for 这类程序流程关键字和 int、float 这些数据类型关键字,但是其只包括基本的 ANSI C++ 类型,用 typedef 或#define 生成的新类型不被标识。
- ☒ 绿色标识的内容是程序注释,即在 /*...*/ 之间和 // 至行尾之间的部分。
- ☒ 灰色的代码是由类向导(Class Wizard, Visual C++ 6.0 的组成部分之一)维护的代码,不建议用户修改,因为修改可能导致类向导的工作不正常,并且不能保证改动最后能被保留下来——类向导完全有可能重写这段代码。

1.5.12 使用 Visual C++ 6.0 编写和运行“Hello, World”程序

介绍了这么多,你是否已经迫不及待地想要用 Visual C++ 6.0 编写和运行一个 C 程序?下面就使用 Visual C++ 6.0 来编写和运行前面提到的“Hello, World”程序。基本步骤如下:

1. 建立项目

第 1 步 打开 Visual C++ 6.0,选择 File (文件)→New (新建)命令,弹出 New (新建)对话框,如图 1.26 所示。切换到 Project (工程)选项卡。

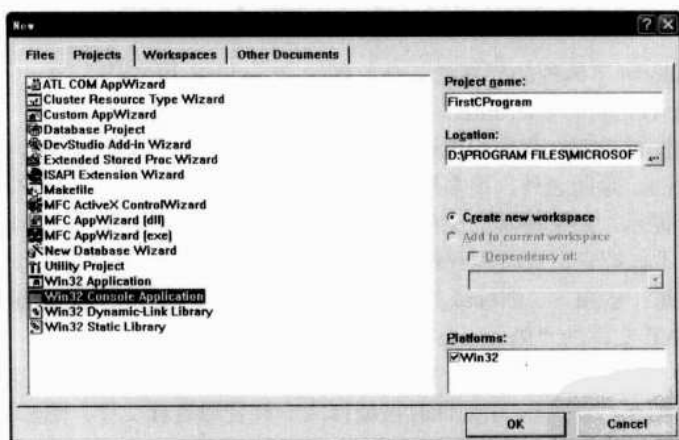




图 1.26 New 对话框

 **提示:** 在对话框左侧的列表框中显示了很多项目类型,一般常用的有 MFC AppWizard(dll) (MFC 动态链接库)、MFC AppWizard(exe) (MFC 可执行程序)、Win32 Application (普通 Windows32 程序)、Windows32 Console Application (Windows32 控制台程序)、Windows32 Dynamic-Link Library (Windows32 动态链接库)。

第2步 选择 Windows32 Console Application (Windows 32 控制台), 右面有两个文本框, 一个是 Location (位置) 文本框, 即项目文件夹的位置。此处, 可以直接输入, 也可以单击其后面的  按钮选取。

 **提示:** Windows32 Console Application (Windows 32 控制台) 是一种很接近于 DOS 程序的 Windows 程序, 没有窗口、没有消息循环, 结构简单。

另一个是 Project name (项目名称) 文本框, 也是项目文件夹的名字 (项目文件夹的实际位置是 Location\Project), 其他的内容使用默认选项, 单击 OK 按钮即可, 然后进入下一设置。此处, 设定项目名称是 FirstCProgram, 位置是 D:\Program Files\Microsoft Visual Studio\MyProjects\。

2. 设置程序向导

完成上述设置后, Visual C++ 6.0 会启动一个称为 Application Wizard (程序向导) 的组件, 如图 1.27 所示。通过向用户询问一些内容来自动产生项目的框架性代码, 用户只需在这个构架中填入实现具体功能的代码即可完成一个程序。Console 程序的程序向导相对比较简单, 其并不需要太多的信息。这里显示的对话框中有 4 个单选按钮, 其意思分别如下:

- ☒ An empty project: 一个空项目, 它不包含任何文件, 所以一般不会使用这一项。
- ☒ A simple application: 一个简单的项目, 它包含一些纯构架性的代码, 生成后可以直接编译成一个“什么都不做”的 EXE 程序, 然后程序员在里面加入需要实现的功能即可完成程序的设计。

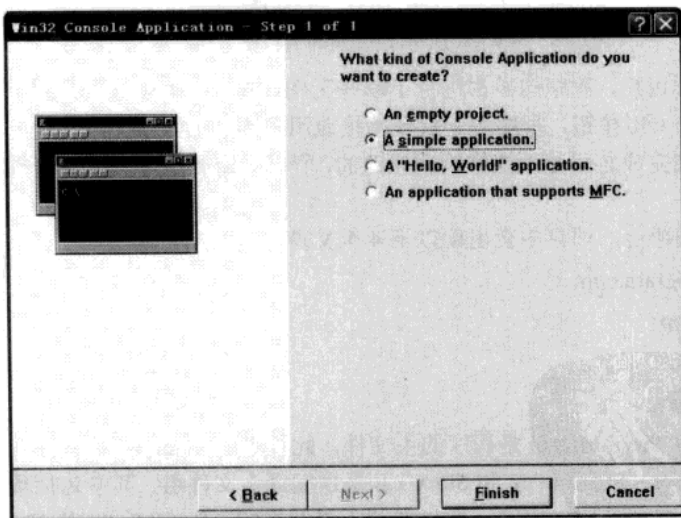


图1.27 程序向导

- ☒ A "Hello, World!" application: 它产生的是一个“Hello, World”程序的源代码, 在 Console 下, 此项没什么用处, 但在其他类型的项目中, 有时会先建立一个这种类型的项目再加以修改得到最终程序, 这比从 simple application 开始要省力得多。



- ☒ An application that supports MFC: 一个支持 MFC 的项目, 和 A simple application 差不多, 但使用它的项目可以使用 MFC 中非界面部分的类, 这是经常使用的选项, 毕竟, 使用 MFC 比纯粹用 API 要方便得多。

此处, 如果选择 A “Hello, World!” application 会直接生成需要的程序, 但是这样不利于了解整个程序的编写过程, 所以此处选择: A simple application 单选按钮, 然后单击 Finish 按钮完成程序向导的设置。之后, 会出现一份报告, 如图 1.28 所示。

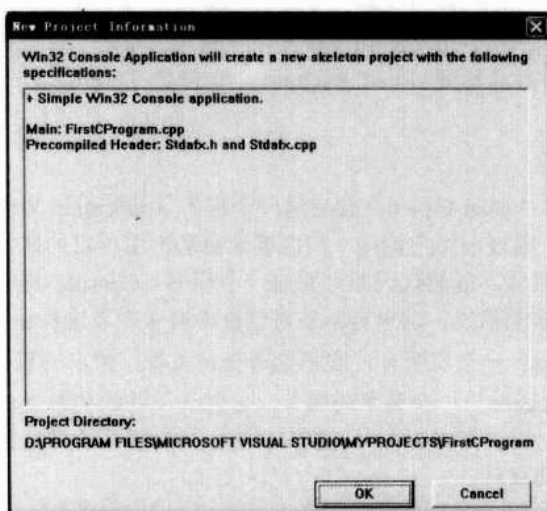


图1.28 New Project Information (新建工程信息) 对话框

此报告告诉用户, 程序向导都进行了哪些工作, 同时, 此处也是给用户最后一次后悔的机会, 一旦单击 OK 按钮, 程序向导就开始按照用户刚才的设置完成工作。要知道, 程序向导进行的工作在完成之后是不能撤销或修改的, 所以一定要认真阅读这份报告来确定刚才的设置是否正确。

单击 OK 按钮后, 项目中会出现以下 4 个文件:

- ☒ FirstCProgram.cpp。
- ☒ StdAfx.cpp。
- ☒ StdAfx.h。
- ☒ ReadMe.txt。

其中, FirstCProgram.cpp 是程序的主文件, 此程序的 main() 函数就在其中, 也是用户编写代码最多的地方; StdAfx.cpp 和 StdAfx.h 是预编译头文件组, 其中包括预编译等内容, 因为此程序不需要修改其中的内容, 所以在此不详细介绍。ReadMe.txt 是 Visual C++ 6.0 产生的一份项目描述文件, 里面解释了项目各个文件的大致内容和意义。

3. 编写自己的代码

到目前为止, 程序实质上是空的, 什么也不做。其实只需要双击 FirstCProgram.cpp 文件, 就会出现其内容文本框。在其中输入 “Hello, World” 程序的代码即可。请读者注意, Visual



C++ 6.0 自动生成的代码中并不包含 `#include <stdio.h>`, 用户必须自己增加此引用。如果没有增加, 程序将无法运行。程序清单如下:

```
// FirstCProgram.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "stdio.h"

int main(int argc, char* argv[])
{
    /*使用 printf() 函数输出 "Hello,World"*/
    printf("Hello,World!\n");
    return 0;
}
```

4. 编译并运行程序

输入代码的工作完成后, 就可以编译和运行程序, 步骤如下:

第 1 步 选择 Build (建立项目) → Compile FirstCProgram.cpp (编译 First CProgram.cpp), 或者按 Ctrl+F7 组合键, 完成对程序的编译。

第 2 步 选择 Build (建立项目) → Buile FirstCProgram.exe (编译 First CProgram.exe), 或者按 F7 键, 完成对程序的连接。

第 3 步 选择 Build (建立项目) → Execute FirstCProgram.exe (编译 Execute First CProgram.exe), 或者按 【Ctrl+F5】 组合键, 执行程序。当然, 此处运行后的程序是 Debug 版的, 如果希望编译 Release 版可以通过前面介绍的方法, 从设置菜单中设置可运行配置 (Set Active Configuration), 设置目前编译配置为 Release 即可。



注意: 也可单击 Build (建立项目) 工具栏中 Compile (编译)、Build (建立项目)、Execute Program (执行程序) 按钮完成程序的编译和运行, 而且大多数情况下都是使用此种方式的。





第 2 章

数据及数据类型



所有的计算机程序，包括 C 程序在内，处理的对象都是数据。数据是程序的必要组成部分，它有常量和变量之分。从外部形态来看，数据包括数值、文字、图像、声音和视频等形式。这些形式多样的数据，在 C 程序中，主要通过整数、实数、字符、字符串等数据类型的方式来表现。

本章就从数据入手，带领读者认识 C 语言中的各种数据类型，并在此基础上介绍 C 语言中的基本数据类型。最后，本章还将介绍 C 语言中用于接受和输出这些类型数据的标准输入/输出函数。这些基础性的内容将始终贯穿在本书后续章节的程序示例中，建议读者仔细体会本章所述内容，并力求熟练掌握。



2.1 数据在计算机中的表示

对于数据,人们并不陌生,日常生活中会遇到各式各样简单或者复杂的数据。本节就从数据的概念入手,介绍数据的概念,C语言中表示数据的符号——字符集和标识符,以及数据在计算机中的存储形式等内容。

2.1.1 数据

数据是对客观事物的符号化表示,是用于表示客观事物未经加工的原始素材,如图形符号、数字、字母等。也可以认为,数据是通过物理观察得来的事实、事件和思想等(统称客体),是关于现实世界中的地方、事件、其他对象或概念的描述。

在计算机科学中,数据是指所有能输入到计算机中并被计算机程序处理的符号的介质的总称,是对输入计算机进行处理、具有一定意义的数字、字母、符号和模拟量等的通称。例如,下面的数字、日期和字母都是数据:

100, 2009-05-01, A

下面的符号也是数据:

♣, @, ☆, ♪

此外,图2.1所示的信号量和图2.2所示的图像也都是数据。

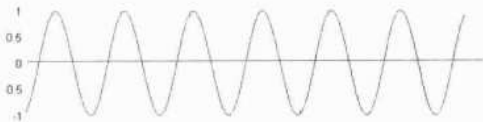


图2.1 信号量数据



图2.2 图像数据

2.1.2 字符集和标识符

任何一门高级语言,都有其基本词汇符号(也称为标识符)和语法规则。程序所处理的数据都是由这些基本词汇符号根据该语言的语法规则组合而成的一个集合。C语言中也规定了其所需的基本字符集和标识符。

1. 字符集

所谓字符集,就是可以使用的字符的集合。满足C语言语法要求的字符集包括以下3种:

- ☑ 英文字母 a~z, A~Z。
- ☑ 阿拉伯数字 0~9。



☐ 特殊符号。

C 语言中可以使用的特殊符号如表 2.1 所示。

表2.1 C语言中可以使用的特殊符号

+	-	*	/	%	=	null
{	}	()	[]	alert
_	`	.	:	?	~	backspace
<	>	&	;	"		CR
!	#	空格	TAB	垂直制表符	换行	换页

2. 标识符

在 C 语言中,标识符是指用来对变量、符号常量、函数、数组、数据类型等数据对象命名的有效序列。简单地说,标识符就是变量、符号常量、函数、数组、数据类型等数据对象的名字。既然是名字,其作用也如同现实生活中的名字一样,起到对数据对象的标识、分辨的作用。

C 语言规定标识符只能由字母、数字和下画线 3 种字符组成,并且第一个字符必须是字母或下画线。下画线“_”起到字母的作用,可用于一个长名字的描述。例如,有一个变量,名字为 `firstschoolname`,识别起来比较困难,如果合理使用下画线,将其写成 `first_school_name`,那么标识符的可读性就大大增强了。例如,下面都是合法的标识符:

```
name、_result、key_board
```

而下面则都不是合法的标识符:

```
1_name      //不能以数字 1 开头  
$100        //包含非法字符$  
key.board   //包含非法字符.
```


3. 标识符类型

标识符包括如下 3 类。

☐ 保留字: 又称关键字,是被 C 语言保留使用的标识符,每一个都有特定含义,不允许用户将其当做变量名使用, C 语言的保留字都用小写英文字母表示,共有 31 个,如表 2.2 所示。

表2.2 C语言的保留字

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	static	struct
switch	typedef	union	unsigned	void	volatile
while					

 **注意：**因为 C 语言是区分大小写的，所以程序中出现的关键字都必须严格按照表 2.2 所示的那样全部采用小写字母。其实，除了关键字，标准库中的函数名（例如 `printf()` 函数）也是用小写字母来书写的。如果在程序中书写大写字母的关键字或标准函数名，例如 `INT`、`Printf` 这样的形式，编译器是不能进行正确识别的。

- ☑ **预定义标识符：**除了保留字外，还有一类具有特殊含义的标识符，其被用作库函数名和预编译命令，此类标识符在 C 语言中称为预定义标识符。一般来说不要将其再定义为其他标识符（用户定义标识符）使用。预定义标识符包括预编译程序命令和 C 编译系统提供的库函数名。其中预编译程序命令有：

```
define undef include ifdef ifndef endif line
```


关于预定义标识符会在第 8 章“预处理”一章中详细介绍。

- ☑ **用户定义标识符：**是程序员根据自己的需要定义的一类标识符，用于标识变量、符号常量、用户定义函数、类型名和文件指针等。

4. 使用标识符注意事项

在使用标识符时必须注意以下几点：

- ☑ **标准 C 不限制标识符的长度，**但其受各种版本的 C 语言编译系统限制，同时也受到具体机器的限制。例如，在某版本 C 中规定标识符前 8 位有效，当两个标识符前 8 位相同时，则被认为是同一个标识符。
- ☑ **在标识符中，大小写是有区别的。**例如 `NAME`、`name`、`Name` 是 3 个不同的标识符。
- ☑ **标识符虽然可由程序员随意定义，但标识符是用于标识某个量的符号。**因此，命名应尽量有相应的意义，以便阅读理解，作到“顾名思义”。
- ☑ **用户定义标识符不能是保留字或预定义标识符。**

 **提示：**一般初学者经常会死记硬背那些标识符、保留字等，实际上不必要这么做，随着编程语法的深入学习，自然就会避免将保留字定义成变量名。因此，编程关键在于掌握基本原理、基本方法和技能，靠死记硬背是不能解决问题的。


2.1.3 数据在计算机中的表示

计算机执行程序的过程中，构成程序的指令和其操作的数据都存储在计算机的某个地方。在程序运行过程中，其存储在计算机的内存中。内存又称主存或随机存储器（RAM）。可以把计算机的 RAM 想象成一组排列好的盒子。每个盒子有以下两种状态：

- ☑ 一种状态下盒子是满的，用 1 来表示。
- ☑ 一种状态下盒子是空的，用 0 来表示。

因此，每一个盒子代表一个二进制数位，即 0 或 1。计算机有时用 `true` 和 `false` 来表示这两种状态，即 1 表示 `true`，0 表示 `false`。每个盒子称为一个二进制位，简称位（bit）。



 **注意：**关于二进制数字，此处不做太多的介绍，如果读者想对其进行深入了解，请查阅相关书籍。此处需要重点了解的是计算机只能处理 1 和 0，不能直接处理数学上用的十进制数，比如 168。程序处理的所有数据和指令，在计算机内部都是由二进制数字构成的。

为了处理方便，位在计算机中都被分成 8 个一组，每 8 个位称为一个字节（byte，单位符号为 B）。为了便于引用特定字节的内容，每个字节分配一个数字标记，第一个字节从 0 开始，第二个字节表示为 1，直到内存中的任意多个字节为止。字节的这种数字标记被称为地址。在内存中，每一个字节都有不同于其他字节的地址，就像宾馆中某个房间对应一个唯一的房间号一样，字节的地址也能唯一地引用计算机内存中的特定字节。

例如，假设地址为 1025 的字节中存储了数字 100，地址为 1026 的字节存储了数字 200，其在内存中的表示如图 2.3 所示。

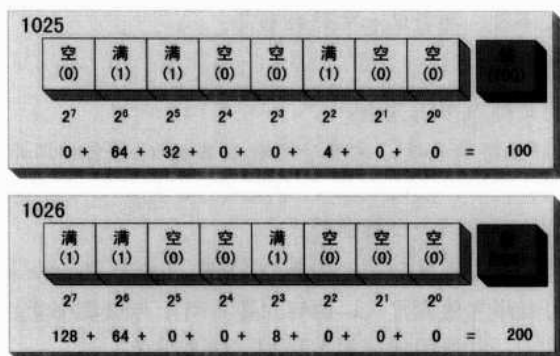


图2.3 内存中的字节

现在，计算机内存的容量一般都很大，所以采用千字节、兆字节甚至吉字节表示，其含义如下：

- ☐ 1 千字节（1KB），表示 1024 个字节。
- ☐ 1 兆字节（1MB），表示 1024 个千字节，即 1,048,576 个字节。
- ☐ 1 吉字节（1GB），表示 1024 个兆字节，即 1,073,741,841 个字节。

没有使用如一千、一百万或十亿这样的数字表示内存容量，是因为内存中的字节是用二进制数表示的，而不是由十进制数表示的。从 0~1023 有 1024 个数字，且 1023 恰好是非常便于使用的 10 位二进制数字，即 11 1111 1111。虽然 1000 是一个非常便于使用和记忆的十进制数，但对于计算机来说却非常不便，其二进制数表示为 11 1110 1000，既不精确也不整齐。所以，用 10 个位来表示千字节，20 个位来表示兆字节，30 个位来表示吉字节是一个非常好的方式。

2.2 数据类型

C 语言包含多种数据类型，这也使其对数据的处理更加丰富。本节就从数据类型的概念入手，介绍 C 语言中的基本数据类型。



2.2.1 数据类型的解释

虽然数据在计算机中都是用二进制数来表示的,但是高级语言对数据的处理更加接近于数学语言。例如,100 和 100.100 在数学上是两种类型的数据,100 是整数,100.100 是小数。此种表示方式也更加接近于人们的生活,所以,计算机高级语言程序中的数据也划分成各种类型。所谓数据类型是指按照数据的性质、表示形式、存储空间的大小、构造特点来划分数据,每个数据都必须有确定的类型。

其实,对于“类型”这个词人们并不陌生。比如,军人可以包括陆军、海军、空军,都可以看成是军人的类型。之所以划分这些类型是因为虽然陆军、海军、空军都属于军人,但是陆军的作战方式、使用的装备、训练方式等方面都与海军和空军有所不同。如果不划分类型,“胡子眉毛一把抓”必将乱了套。

计算机中的数据也一样,各种数据都有其自身的特点。例如,对于 100 和 'A' 这两个数据,人们在理解时肯定是看成两种类型来考虑的,而不是当做一种数据来思考的;程序设计语言一般也是对其采用不同的编码和处理方式。所以,划分数据类型,既有利于人们的理解,又便于计算机对其进行存储和处理。

2.2.2 C 语言中的数据类型

各种程序设计语言对于数据类型的划分都不尽相同,对于 C 语言来说,其数据类型的分类方式如图 2.4 所示,各种数据类型的解释如下。

1. 基本类型

基本类型包括整型、实型、字符型和枚举型。其最主要的特点是,其值不可以再分解为其他类型,就像化学中介绍的原子一样不可再分。

2. 构造数据类型

构造数据类型包括数组类型、结构体类型、共用体(联合)类型,根据已定义的一个或多个数据类型用构造的方法来定义的。也就是说,一个构造类型的值可以分解成若干个“成员”或“元素”。每个“成员”都是一个基本数据类型或又是一个构造类型,就像化学中介绍的分子,又可以分解为若干个原子。

3. 指针类型

指针是一种特殊的、具有重要作用的数据类型,其值用来表示某个变量在内存中的地址。虽然指针变量的取值类似于整型量,但其和整型是完全不同的两种类型,使用时要特别注意。关于指针类型会在第 7 章“指针”中详细介绍。

4. 空类型

在调用函数值时,通常应向调用者返回一个函数值。这个返回的函数值具有一定的数据类型,应在函数定义及函数说明中给予说明,例如:

```
y=sqrt(x);
```

此语句完成对 x 求平方根并赋值给 y,也就是说,函数 sqrt() 要返回一个整型或者实型的值赋给变量 y。

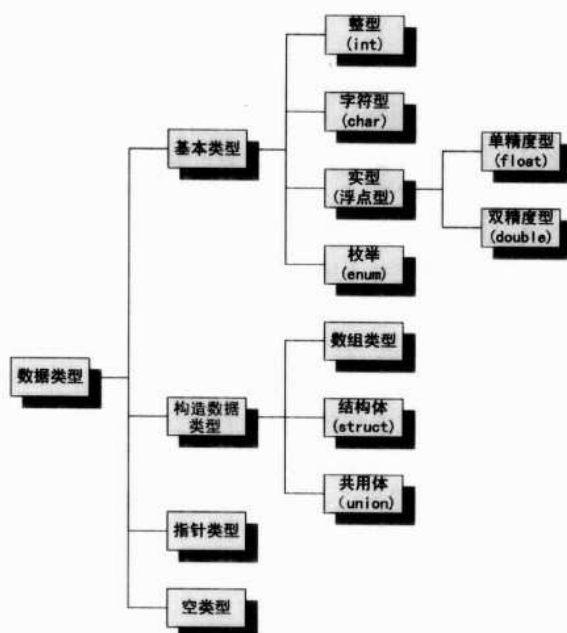



图2.4 数据类型

但是，也有一类函数调用后并不需要向调用者返回函数值，这种函数可以定义为“空类型”，其类型说明符为 `void`。对此会在第 5 章“函数”中详细介绍。

 注意：本章只介绍基本数据类型中的整型、字符型和实型，其他类型会在后续章节中介绍。

2.3 常量

常量是在程序运行中数值不变的数据量。因其本身已经包含了数据类型定义，所以在使用时不必事先进行声明。本节就来介绍 C 语言中的常量。

2.3.1 整型常量

整型常量简称整常数或整数。在 C 语言中，可用十进制数、八进制数、十六进制数来表示。

1. 十进制整数

十进制整数没有前缀（加在数据前用来区分各种进制的数字或字符）。其数码为 0~9。不能以 0 为开头。包括有符号数和无符号数。

例如，下面的数都是合法的十进制整数：

```
0、-1、255、-512、65535
```

而下面的数都不是合法的十进制整数：

```
08      // 不能有前导 0
100A    // 含有非十进制数码 A
```



```
655 35    //含有非十进制数码空格
65,535    //含有非十进制数码逗号
```

2. 八进制整数

八进制整数以 0 作为前缀,也就是说,必须以 0 为开头。其数码为 0~7。其通常是无符号数。

例如,下面的数都是合法的八进制整数:

```
01        //十进制数为 1
0377      //十进制数为 255
0177777   //十进制数为 65535
```

而下面的数不是合法的十进制整数:

```
255       //无前缀 0
-01       //出现了负号
028       //包含了非八进制数码 8
```

3. 十六进制整数

十六进制整数的前缀为 0X 或 0x。其数码为 0~9、A~F 或 a~f。其中 A~F 或 a~f 表示十进制数的 10~15。

例如,下面的数都是合法的十六进制整数:

```
0X1       //十进制数为 1
0XFF      //十进制数为 255
0XFFFF    //十进制数为 65535
```

而下面的数不是合法的十六进制整数:

```
6A        //无前缀 0X 或 0x
0X1Y      //含有非十六进制数码
```

在 16 位字长的机器(16 位字长是指该计算机中央处理器(CPU)能同时处理 16 位二进制信息,其反映了该计算机的处理能力)上,基本整型的长度也为 16 位,因此表示的数的范围也是有限定的。

- ☑ 对于无符号数,其最小数为所有位上是 0 的数,最大数为所有位上全部是 1 的数。
- ☑ 对于有符号数,其最小数为其他 15 位上是 0 再加上符号位为 1 的数,最大数为其他 15 位上都是 1 再加上符号位为 0 的数。

十进制无符号整数的范围为 0~65535,有符号数为-32768~+32767。八进制无符号数的表示范围为 0~0177777。十六进制无符号数的表示范围为 0X0~0XFFFF 或 0x0~0xFFFF。如果使用的数超过了上述范围,就必须用长整数来表示。从而引入了整数后缀,对于长整数是用后缀“L”或“l”来表示的。也就是说,在其后加“L”或“l”,表示此数是长整数。在 16 位字长的机器上,其表示范围为-2147483648~+2147483647,例如:

☑ 十进制长整数:

```
100L      //十进制数为 100
65536L    //十进制数为 65536
```

☑ 八进制长整数:

```
0144L     //十进制数为 100
0200000L  //十进制数为 65536
```



▣ 十六进制长整数:

```
0X64L          //十进制数为 100
0X10000L       //十进制数为 65536
```

从上面的例子中可以看出,长整数 100L 和基本整数 100 在数值上并无区别。但对 100L, 因为是长整型量, C 编译系统将其分配 4 个字节的存储空间; 而对 100, 因为基本整型, 只分配 2 个字节的存储空间。也就是说其在计算机中的表示是不同的。因此在运算和输出格式上要加以注意, 避免出错。

无符号数也可用后缀表示, 整型常数的无符号数的后缀为 “U” 或 “u”, 例如:

```
100u, 0X10000U //无符号整数
0X64Lu          //无符号长整数
```

以上均属于无符号数。

提示: 前缀、后缀可同时使用以表示各种类型的数。例如, 0X64Lu 表示十六进制无符号长整数 64, 其十进制数为 100。

2.3.2 实型常量

实型常量也称为实数或者浮点数。类似与数学中学习的小数。提出浮点数, 是由于程序经常要处理一些非常大或者非常小的数, 而用整数无法对其表示。在 C 语言中, 实数只采用十进制数, 不能像整数一样, 用八进制数或十六进制数表示。例如, 下面的实数都是合法的:

```
3.14、-1.11、65535.0
```

而下面的实数都是非法的:

```
03.14          //没有八进制的实数
0X100A.123     //没有十六进制的实数
```

实数有两种形式: 十进制小数形式和指数形式。

1. 十进制数形式

由数码 0~9 和小数点, 以及可能的正、负号组成。

例如:

```
0.0、3.1415926、.123、100.、-65535.0123
```

上面都是用十进制数形式表示的合法实数。

注意: 实数必须有小数点。但小数点前后都可以没有数码。

2. 指数形式

由尾数部分、指数标志 “e” 或 “E”, 以及指数部分 (只能为整数, 可以带符号) 组成。其一般形式为:

```
a E n 或 a e n    //a 为十进制数, n 为十进制整数
```

其值为: $a \times 10^n$ 。例如, 下面都是用指数形式表示的合法实数:

```
3.14E6          //等于  $3.14 \times 10^6$ 
7.25E-4         //等于  $7.25 \times 10^{-4}$ 
-6.28E-6        //等于  $-6.28 \times 10^{-6}$ 
```

此形式中必须包含尾数部分，且必须为合法的实数。指数部分只能为整数，可以带符号（+或者-），并且，如果有指数标志，则必须有指数部分。例如，以下都不是合法的实数：

```
123E2    //无小数点
E2        //无尾数部分
1.23E    //无指数部分
```

标准 C 允许实数使用后缀“f”或“F”，即表示该数为实数。使用后缀可以省略小数点，例如，123.可用后缀形式表示为 123f。一般的计算机系统中，一个单精度实数在内存中占用 4 个字节，具有 6~7 位有效数字，取值范围为 $10^{-38} \sim 10^{38}$ 。双精度实数占用 8 个字节，具有 15~16 位有效数字。

2.3.3 字符常量

字符常量是用一对单引号括起来的一个字符。例如，下面都是合法的字符常量：

```
'x'、'Y'、'1'、'='、'?'、'Y'
```

而下面都不是合法的字符常量：

```
"x"、'abc'
```

C 语言中一个字符常量占用一个字节，但该字节中存放的并非此字符本身，而是该字符所在机器采用的字符集的代码，是一个整型数。大多数系统采用 ASCII 代码字符集，其中'a'的 ASCII 值为 97，'A'的 ASCII 值为 65，'0'的 ASCII 值为 48（字符的 ASCII 值参见附录 II）。因其存储的是一个数字，故其可以像整数一样参与数值运算。

在 C 语言中，字符常量具有以下特点：

- ☑ 字符常量只能用单引号括起来，不能使用双引号或其他形式。
- ☑ 字符常量只能是单个字符，不能是多个字符。
- ☑ 字符可以是字符集中任意字符。但数字被定义为字符型之后就不能参与数值运算。如 '10'和 10 是不同的。'10'是字符常量，不能参与运算。

C 语言还使用了一种特殊的字符常量，即转义字符。它以反斜线“\”开头，后跟一个或几个字符。之所以叫转义字符，是因为其具有特定的含义，不同于字符原有的意义，故称为“转义”字符。转义字符常用于表示 ASCII 码字符集内的控制代码，例如，'\n'是一个转义字符，表示“回车换行”，其实际上是一个字符，ASCII 码值为 10。常用的转义字符如表 2.3 所示。

表2.3 常用转义字符及其含义

转义字符	转义字符的意义	ASCII 代码
\n	回车换行	10
\t	横向跳到下一制表位置	9
\b	退格	8
\r	回车	13
\f	走纸换页	12
\\	反斜线符“\”	92
'\'	单引号符	39



(续表)

转义字符	转义字符的意义	ASCII 代码
\"	双引号符	34
\a	鸣铃	7
\ddd	1~3 位八进制数所代表的字符	
\xhh	1~2 位十六进制数所代表的字符	

其实, C 语言字符集中的任何一个字符都可用转义字符来表示。表 2.3 所示的 \ddd 和 \xhh 正是为此而提出的。ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。例如, \141 和 \x61 都表示字母“a”, \134 和 \X5C 都表示反斜线。

2.3.4 字符串常量

字符串常量是由一对双引号 (") 括起的字符序列。所谓字符序列, 指由一个或多个合法字符构成的序列。例如, 下面都是合法的字符串常量:

```
"C"、"Welcome"、"This is a C program"、"$100.5"
```

字符串常量和字符常量, 特别是单个字符的字符串常量和字符常量很相似, 但它们之间是存在差异的。字符常量和字符串常量的区别如下:

- ☑ 字符常量由单引号 (') 括起来, 字符串常量由双引号 (") 括起来。
- ☑ 字符常量只能是单个字符, 字符串常量则可以包含一个或多个字符。
- ☑ 可以把一个字符常量赋予一个字符变量, 但不能把一个字符串常量赋予一个字符变量。在 C 语言中没有相应的字符串变量, 一般是用一个字符数组来存放一个字符串常量, 将在第 6 章“数组与字符串”一章中详细介绍。
- ☑ 字符常量占一个字节的内存空间。字符串常量占的内存字节数等于字符串中字节数加 1, 增加的一个字节中存放字符 '\0' (ASCII 码为 0) 这是字符串结束的标志, 从计算机存储上来看, 长度为 n 字节的字符串, 占用了 $n+1$ 个字节的存储空间。

例如, 字符串 "Hello,World" 在内存中所占的字节如图 2.5 所示。




图2.5 字符串常量存放图

字符常量 'x' 和字符串常量 "x" 虽然都只有一个字符, 但在内存中的情况是不同的。其中: 'x' 在内存中占一个字节, 如图 2.6 (a) 所示; "x" 在内存中占两个字节, 如图 2.6 (b) 所示。



图2.6 字符常量和字符串常量


注意：字符串常量中的字符也可以是转义字符。例如，`"\\Hello\\"`，其值包含双引号（`"`）和 `Hello`。

2.3.5 符号常量

在 C 语言中，常量可以用标识符来命名，即符号常量。例如，下面求圆形周长和面积的程序，就用到了符号常量。

```
#define PI 3.1415926
void main()
{
    int r;           //定义变量 r 存放圆半径
    float girth, area; //定义变量 girth、area 存放圆周长和面积
    girth=2*PI*r;    //计算圆周长
    area=PI*r*r;     //计算圆面积
}
```

此程序首先用 `#define` 定义了一个符号常量 `PI`，其值为 `3.1415926`，此后在此文件中出现的 `PI` 都等价于 `3.1415926`。使用 `PI` 和求圆周长和面积的公式求出圆的周长和面积。关于 `#define` 会在第 8 章“预处理”中详细介绍。

注意：符号常量的值在其作用域（上面的程序为主函数）内不能改变，也不能再被赋值。

例如，如下在主函数中再对 `PI` 赋值是错误的。

```
PI=3.1415927;
```

为了与变量区分，一般情况下，符号常量名用大写字母。其在使用前必须先定义，一般形式为：

```
#define 符号常量名 常量
```

使用符号常量有如下好处：

☑ 含义清楚

例如上面的程序中，看到符号常量 `PI` 就会联想到圆周率“ π ”，可谓是“顾名思义”。因为在检查程序时会搞不清楚每个常数究竟代表什么。应尽量使用“见名知意”的变量名和符号常量。

☑ 能做到“一改全改”

例如上面的程序中，如果对于圆周率只要求保留到小数点后两位，即 `3.14`。使用上面的程序的话只需要修改一处，也就是将第一行替换为：

```
#define PI 3.14
```

而如果没有采用符号常量的话，对上面求圆周长和面积的程序则需要修改两处。对于一些复杂的程序，其中的某些常量可能被多次使用，在这些常量值需要修改时，采用符号常量的形式将会使修改非常高效和方便。而且，在程序的开头处修改总比在程序中找到后再改要方便得多。

无论是符号常量还是其他的常量，其值都不能被任意改变。而程序中用到的很多数据都不是一成不变的，这就引出了下一节要介绍的另外一种数据——变量。



2.4 变 量

变量是在程序运行中数值变化的数据量。其必须有一个名字、使用前必须先定义、在内存中占有一定的存储空间。本节介绍 C 语言中各种类型的变量。

2.4.1 给变量命名

C 语言中每个变量都必须有一个变量名,此变量名必须是一个合法的标识符。准确地说,其应该是一个合法的用户定义标识符。

也就是说,变量名由字母或下画线“_”开始,后跟一定数量的字母、数字或者下画线。例如,下面都是合法的变量名:

```
average           //表示平均值
number_3rd        //表示编号为 3 的量
number_of_students //表示学生总数
```

而下面都是不合法的变量名:

```
3rd_entry         //以数字开始
all$done          //包含非法字符“$”
the end           //包含非法字符空格
```

在对变量命名时,需要注意以下几点:

- ☑ 命名时应注意区分大小写,并且尽量避免只是大小写上有区别的变量名。例如,变量名为 jack、Jack、JACK 的 3 个变量是不同的变量。虽然在一个程序中命名 3 个这样的变量是允许的。但是,为了避免混淆,应该根据变量所代表的含义使用不同的变量名。
- ☑ 不建议使用以下画线开头的变量名,因为此类名称通常是保留给内部和系统的名字。例如, `_dos_getdrive` 和 `_chmod` 这两个名称一般只是在 DOS 环境中才使用的变量和函数。而实际编程中可能运行在 Windows、UNIX、Linux 操作系统上,所以建议采用 `getdrive` 和 `chmod` 这样的名称。
- ☑ 不能使用 C 语言保留字或者预定义标识符作为变量名。例如,下面的变量名是非法的:

```
int               //C 语言保留字
include           //预定义标识符
```

- ☑ 避免使用类似的变量名。例如,下面是一些不好的变量名:

```
total            //记录的总数
totals            //全部记录的总数
```

上面的两个变量名很容易引起混淆,对其好的命名方式如下:

```
envry_total      //某条记录的总数
all_total        //全部记录的总数
```

2.4.2 变量定义

在 C 语言中,在使用变量之前必须首先定义此变量。定义变量的目的在于:


- ☐ 定义了变量的名称。
- ☐ 定义了变量属于哪种数据类型。
- ☐ 给程序员提供关于此变量的描述信息。

变量定义的一般形式是：

```
变量类型 变量名;           //注释
```

其中，变量类型可以是 C 语言提供的所有的变量类型，包括基本类型和其他类型。变量名是一个能反映此变量内容的合法标识符。注释是关于这个变量的一些解释性文字，能够增强程序的可读性和易读性。例如，下面是一个变量定义：

```
int result;                 //存储结果的变量
```

 **注意：**注释并不是 C 语言语法规规定定义变量时必须要有有的，但是作为一种好的编程风格，建议在定义变量时使用注释。

其中，关键字 `int` 说明此变量是一个整型变量。变量的名称是 `result`，具备一些英文常识的人看到这个名字就能知道这个变量的作用。分号（`;`）是语句结束的标志。而“`//`”号后面的注释对变量作用进一步解释。

在定义同一类型的多个变量时，C 语言允许在一行或多行上列出变量名，并用逗号将各个变量名分隔。例如，下面是运用此规则的变量定义：

```
int age,weight,height;     //存储年龄、体重、身高的 3 个变量
```

其等价于如下定义：

```
int age;                   //存储年龄的变量
int weight;                //存储体重的变量
int height;                //存储身高的变量
```

2.4.3 变量名与变量的值

一个变量总是拥有一个变量名，其中可以存放一个值，且该值可以被改变，这也是称其为变量的原因。变量名的作用是对应一个内存位置，而变量值是对应此位置中的具体数值。例如，可以将变量名看成一个教室的编号，而将变量值看成此教室中某时刻的人数。图 2.7 所示为某时刻两个教室中的学生数。

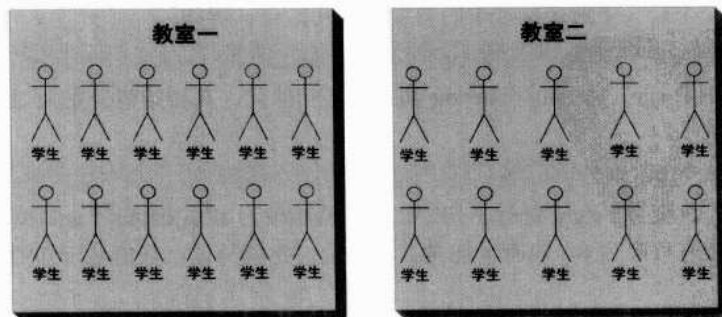


图2.7 教室人数图



上述情况可以用变量表示如下：

```
int classroom_1=12;    //定义教室一中的人数
int classroom_2=10;    //定义教室二中的人数
```

其中，第一个定义定义了名字为 `classroom_1` 的 `int` 型变量用来存放教室一中的人数，当前的人数是 12。第二个定义定义了名字为 `classroom_2` 的 `int` 型变量用来存放教室二中的人数，当前的人数为 10。变量在内存中的存储如图 2.8 所示。

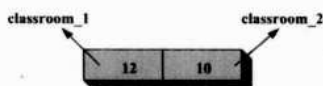


图2.8 变量在内存中的存储

上面的两个定义还包含了一些新的内容，就是下一小节要介绍的变量初始化。

2.4.4 变量初始化

在程序开始执行时，一些变量的值会自动设置为 0，此数值称为初始值。例如，在 Windows 系统下编写的 C 程序中，`int` 型的变量会自动设置为 0，但大多数变量都不会。这样就会产生人们无法预知变量初始值是什么的问题。根据操作系统和编译器的不同，产生不同的或许有意义，又或许没有意义的值。

如果希望变量有一个初始值的话，可以使用一种简便的办法，就是在变量声明中加入初始值。这种方式就称为变量被初始化，其一般形式是：

```
变量类型 变量名=初始值;    //注释
```

其中，“=”称为赋值运算符，表示将其后边的初始值放入以变量名命名的变量中。初始值可以是如下形式：

- ▣ 常量，例如 1、1.23、'a'。
- ▣ 由常量组成的表达式（表达式将在后续章节中详细介绍），例如 $1+2+3$ 、 $1.23-1$ 、'a'+b'。
- ▣ 常量和前面被初始化的变量组成的表达式，例如 $x+1$ ，其中 x 是一个变量并且已经被初始化。

例如，下面是对 `int` 型数据 `age` 进行初始化：

```
int age=20    //定义存放年龄的变量并初始化为 20
```

可以在同一个定义中对任意数量的变量进行初始化，例如：

```
int age=20,weight=60,height=170;
```

此定义对变量 `age`、`weight` 和 `height` 都进行了初始化，其初始值分别为 20、60 和 170。再来看下面的例子：

```
int age,weight,height=170;
```

此定义只是将变量 `height` 进行了初始化，其初始值为 170；而变量 `age` 和 `weight` 并没有被初始化，其值有可能为 0，也有可能是其他数值，也就是说，其值是未知的。

2.4.5 赋值

变量除了通过初始化的方式获得值外，还可以通过赋值的方式获得值。其一般形式是：

```
变量名=变量值;
```

其中,变量必须在赋值之前进行定义。变量值可以是一个常量或者一个表达式。符号“=”称为赋值号,而不是等号。例如,下面都是合法的赋值语句:

```
age=20;           //变量值为常量
area=PI*r*r;      //变量值为表达式
```

关于赋值运算符将在后续章节中详细介绍。

2.4.6 整型变量

整型变量就是存放整数的变量,整数就是不含小数点的数。例如,100、-88就属于整数。

1. 整型变量在内存中的存放形式

整数在内存中是以二进制的形式存放的,例如,下面定义一个整型变量x:

```
int x=100;        //定义整型变量x并初始化为100
```

一般情况下,整型变量在内存中占用2个字节,也就是16位。图2.9(a)是变量x存放数据的示意图;图2.9(b)所示是变量x在内存中实际存放的形式。

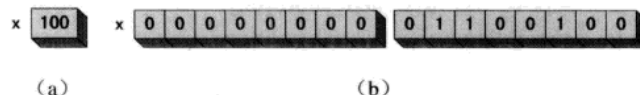


图2.9 变量存放形式图

实际上,在计算机中,数值是以补码的形式表示的。此处首选需要解释一下原码,将任意进制的数转换为二进制数,此二进制数就是原来数的原码。相对于原码,补码的表示规则如下:

- ☑ 正数的补码和原码相同[图2.9(b)既是变量x的原码形式,又是其补码形式]。
- ☑ 负数的补码,先求该数的绝对值的原码,再将其按位取反再加1。

例如,下面求数字-123的补码:

- ☑ 先求其绝对值123的原码,如图2.10(a)所示。
- ☑ 对求得的原码按位取反,如图2.10(b)所示。
- ☑ 对取反后的数加1,得到数字-123的补码,如图2.10(c)所示。

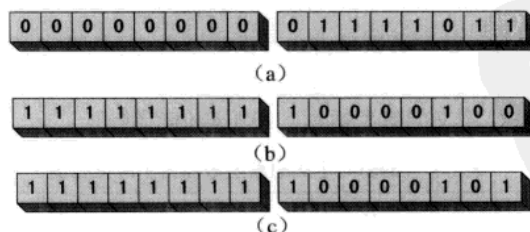


图2.10 求-123的补码

比较图2.9(b)和图2.10(c),可以看出,在整数的16位中,左面的第一位是表示符号的。该位为0,表示此数为正数;为1,表示此数为负数。



2. 整型变量的分类

整型变量可以根据其数值范围和有无符号分为如下 4 种类型：

- ☑ 基本型：类型说明符为 `int`。
- ☑ 短整型：类型说明符为 `short int` 或 `short`。
- ☑ 长整型：类型说明符为 `long int` 或 `long`。
- ☑ 无符号型：类型说明符为 `unsigned`。

其中，无符号型又可分为以下 3 种类型：

- ☑ 无符号基本型：类型说明符为 `unsigned int` 或 `unsigned`。
- ☑ 无符号短整型：类型说明符为 `unsigned short`。
- ☑ 无符号长整型：类型说明符为 `unsigned long`。

也就是说，C 语言中的整型变量可以分为如下 6 种类型。

- ☑ 有符号基本整型，类型说明符为 `[signed] int`。
- ☑ 无符号基本整型，类型说明符为 `unsigned int`。
- ☑ 有符号短整型，类型说明符为 `[signed] short [int]`。
- ☑ 无符号短整型，类型说明符为 `unsigned short [int]`。
- ☑ 有符号长整型，类型说明符为 `[signed] long [int]`。
- ☑ 无符号长整型，类型说明符为 `unsigned long [int]`。

其中，[] 内的部分可以省略。各种无符号类型量所占的内存空间字节数与相应的有符号类型量相同。但由于省去了符号位，故不能表示负数。而且，正是因为无符号数省去了符号位，使其可以存放的正整数的范围比有符号数扩大了一倍。例如，定义两个基本整型变量如下：

```
int x;           //有符号基本整型变量
unsigned int y;  //无符号基本整型变量
```

变量 `x` 的取值范围为 $-32\,768 \sim 32\,767$ ，其最大值 $32\,767$ 在内存中的表示如图 2.11 (a) 所示。而变量 `y` 的取值范围为 $0 \sim 65\,535$ ，其最大值 $65\,535$ 在内存中的表示如图 2.11 (b) 所示。

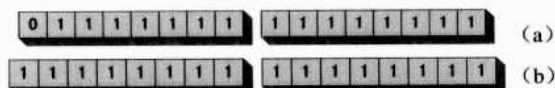


图2.11 基本整型的最大值

标准 C 并没有规定各种整型数据所占内存的字节数，只是规定了各种整型数据的最小存储空间大小。其规定 `short` 类型和基本类型最少占用 2 个字节，而 `long` 类型最少占用 4 个字节。表 2.4 列出了 ANSI C 定义的整型数据的最小取值范围。

表2.4 ANSI C定义的整型数据最小取值范围

类型说明符	字节数	数的范围
<code>int</code>	2	$-32\,768 \sim 32\,767$ 即 $-2^{15} \sim (2^{15}-1)$
<code>unsigned int</code>	2	$0 \sim 65\,535$ 即 $0 \sim (2^{16}-1)$
<code>short int</code>	2	$-32\,768 \sim 32\,767$ 即 $-2^{15} \sim (2^{15}-1)$

(续表)

类型说明符	字节数	数的范围
unsigned short int	2	0~65 535 即 $0\sim(2^{16}-1)$
long int	4	-2 147 483 648~2 147 483 647 即 $-2^{31}\sim(2^{31}-1)$
unsigned long int	4	0~4 294 967 295 即 $0\sim(2^{32}-1)$

例如，将数据 32 767 定义为各种整型形式，在内存中表示（计算机字长为 16 位）如图 2.12 所示。

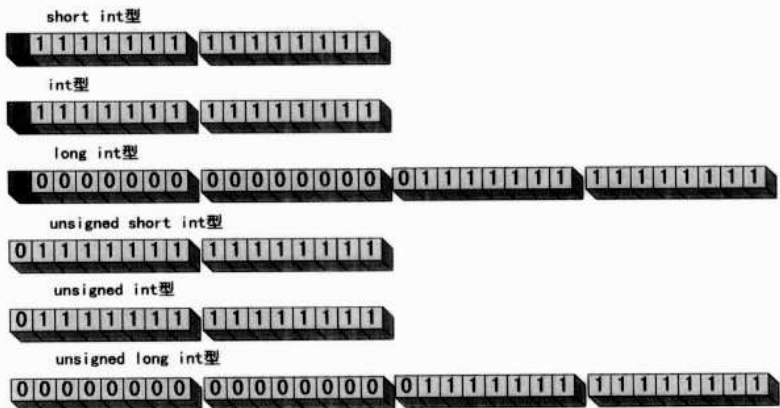



图2.12 数据32 767在内存中的各种形式表示

而且，标准 C 还要求 long 型数据的长度不短于 int 型和 short 型，int 型不短于 short 型。此规则可用下式表示：

$$\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$$

其中，sizeof 为求变量占用内存空间的运算符。

 **提示：**int 类型是最适合计算机系统处理的整型类型，它具有和 CPU 寄存器相对应的空间大小和位格式。

3. 整型变量的溢出

当变量的值超出其数据类型所能存储的值时，就发生了溢出。变量所能存储的值的范围取决于具体的计算机。

对于整数来说，C 语言不提供溢出的任何警告或提示。其只是简单地给出不正确的结果。溢出通常产生一个负数。对于有符号整数来说，如果对其最大值再加 1，结果将变为其最小值。这有点类似于在环形跑道上跑步，终点又是起点。下面的程序清单给基本整型数据 32767 加 1，看看会出现什么结果？

```
#include "stdio.h"
void main()
{
    int x,y;
    x=32767;
```



```

    y=x+1;
    printf("%d%d",x,y);
}

```

此程序在 16 位机下的运行结果将是：

32767, -32768

变量 x 和变量 y 在内存中的存放如图 2.13 所示。

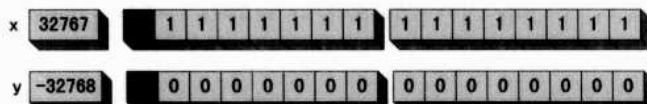


图2.13 变量的内存存放图

由图 2.13 可以看出，变量 x 是一个正整数，其最高位为 0，表示正数，后面 15 位全是 1，用十进制数表示是 32 767。给 x 加 1 后得到变量 y，其最高位是 1，因为是一个有符号数，此位表示符号，此处表示负数，后面 15 位全是 0，是十进制数 -32 768 的补码表示。而不是预期结果 32 768。要达到预期效果，需要将变量 x 和 y 定义为 long 型，并按 %ld 的形式输出。

注意：要根据变量的取值范围定义其数据类型防止溢出，但并不是说要将变量全部定义成成长数据类型（如 long 型）。而且，再长的数据类型都有其范围，要根据程序的逻辑关系调整程序不产生溢出，而不是一味地加长数据类型。

2.4.7 实型变量

实型变量又称为浮点型变量，是其中存放浮点数的变量。浮点数就是带有小数点的数。例如，100、23、-22.8 都属于浮点数。

1. 实型变量在内存中的存放形式

浮点数一般占 4 个字节（32 位）的内存空间。与整数存放方式不同，浮点数是按指数形式存储的。操作系统将一个实数分成小数部分和指数部分。如果用十进制符号表示，其小数部分具有固定的位数，指数部分是以 10 为底的幂。其一般形式为：

小数部分 E (e) 指数部分

其中，E (e) 表示指数 (exponent) 的意思。

例如，下面定义一个单浮点型变量 f：

```
float f=1.23456789 //定义一个浮点型变量并初始化
```

其在内存中的存放形式如图 2.14 所示。

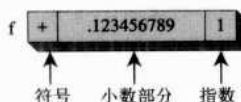


图2.14 浮点数在内存中的表示

注意：图 2.14 中是用十进制数来示意的，实际在计算机中是用二进制数来表示小数部分并用 2 的幂次来表示指数部分的。

至于在这 4 个字节（也就是 32 位）中，用多少位来表示小数部分，多少位来表示指数部分，标准 C 并没有具体规定，而是由编译器来决定的。小数部分占的位数越多，数的有效数字越多，精度就越高。指数部分占的位数越多，则能表示的数值范围越大。

2. 实型变量的分类


实型变量全部是有符号的，可以分为以下 3 种类型：

- ☐ 单精度型，类型说明符为 float。
- ☐ 双精度，类型说明符为 double。
- ☐ 长双精度，类型说明符为 long double。

标准 C 并没有明确规定各种浮点变量占用的内存和值的范围，是由机器及编译器决定的。对于某些编译器来说，long double 类型与 double 类型完全相同。表 2.5 所示是典型的 C 语言编译器对于各种类型的实型变量的占用内存和取值范围。

表2.5 典型C语言编译器中的实型变量

类型说明符	比特数（字节数）	有效数字	数的范围
float	32（4）	6~7	$10^{-37} \sim 10^{38}$
double	64（8）	15~16	$10^{-307} \sim 10^{308}$
long double	128（16）	18~19	$10^{-4931} \sim 10^{4932}$

 注意：表 2.5 中的有效数字并不是一个固定值，是因为计算机中的浮点数是用二进制形式存储的，以单精度浮点数为例，如果其有效数字用二进制数表示是 23 位，其并不完全对应 7 位的十进制数。

3. 实型数据的舍入误差

由于实型变量是由有限的存储单元组成的，因此能提供的有效数字总是有限的，在其有效位以外的数字将被舍弃，因此会产生一些误差，将其称为舍入误差。例如，将一个无限循环小数赋值给一个实型变量，就会产生这种误差。此程序的清单如下所示：

```
#include "stdio.h"
void main()
{
    float x;           //定义单精度实型变量 x
    x=10.0/3;          //将无限循环小数赋值给变量 x
    printf("%f\n",x);  //输出变量 x 的值
}
```

此程序在 Visual C++ 6.0 编译环境下的运行结果如图 2.15 所示。

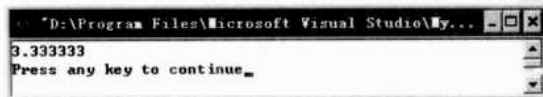


图2.15 输出结果图

从程序的输出结果中可以看出，由于单精度浮点数的有效数字位只有 7 位，所以对于 10.0/3 运算后的无限循环小数只输出了前 7 个有效数位。



注意：要防止对两个量级差别很大的数（某个值特别大，某个值特别小）执行加法或减法运算，因为舍入误差会产生与理论值不同的结果。比如，给 12 345 678.0 加上 0.000 123，其值应该等于 12 345 678.000 123，将其转换为具有 7 位精度的浮点数，将变为 0.1 234 567E+7，对它来说，第 7 位后面的数字是无意义的。

2.4.8 字符变量

字符变量用来存储单个字符，其类型说明符是 `char`。字符变量类型定义的格式和书写规则都与整型变量相同。例如：

```
char c; //定义一个字符变量
```

每个字符变量被分配一个字节的内存空间，因此只能存放一个字符。在内存中，存放的并不是该字符本身，而是对应的 ASCII 码值。字符的 ASCII 码值请参见附录 III。其在内存中的存放形式和整型变量是相同的。例如，将字符 'a' 赋值给上面定义的字符变量 `c`：

```
c = 'a';
```

因为 'a' 的 ASCII 码值是 97，所以字符变量 `c` 在内存中的表示如图 2.16 所示。



图2.16 字符在内存中的存放形式

在 C 语言中，可以将字符变量看成是整型量。允许对整型变量赋以字符值，也允许对字符变量赋以整型值。在输出时，允许把字符变量按整型量输出，也允许把整型量按字符量输出。

可以将存储在字符变量中的整数解释成一个有符号的值，也可以解释成无符号的值，如何解释完全取决于编译器。有符号整数允许存储正数，也允许存储负数。其取值范围是 -128~127。无符号整数只允许存放正数，其取值范围为 0~255。

不过，这两种类型对应的位模式是相同的，即从 00000000~11111111。只不过对有符号值来说，最左边的位是符号位，所以最小数是 10000000，即 -128，最大数是 01111111，即 127。对无符号数来说，全部位都是数据位，所以最小数是 00000000，即 0，最大数是 11111111，即 255。所以从字符代码（即位模式）的角度上来看，字符型是否具有符号并不重要。

前面多次出现了含有 `printf()` 形式的语句，这其实是 C 标准库中用于输出数据的函数。C 标准库提供了多种用于数据输入/输出的函数，本章后面的内容就来详细介绍 C 语言中的标准输入/输出函数。

2.5 数据的输入/输出

程序需要能够与外界交互，也就是用户和计算机之间进行交互。因此，任何一门编程语言必须解决人机交互问题，才能有实际意义。而人机交互问题最简单的形式就是通过对数据的输入/输出来实现的。一般情况下，C 语言中数据的输入/输出是通过标准输入/输出函数来实现的。本节将从输入/输出的概念入手，对 C 语言中的各种标准输入/输出函数进行介绍。

2.5.1 什么是输入/输出

从输入设备向计算机传入数据的行为称为“输入”。所谓输入设备，是人或外部设备与计算机进行交互的一种装置，用于把原始数据和处理这些数据的程序输入到计算机中。包括键盘、鼠标、摄像头、扫描仪、光笔、手写输入板、游戏杆、语音输入装置等。

从计算机向外部输出设备传出数据的行为称为“输出”。所谓输出设备，是人与计算机交互的一种部件，用于数据的输出。其将各种计算结果数据或信息以数字、字符、图像、声音等形式表示出来。包括显示器、打印机、绘图仪、影像输出系统、语音输出系统、磁记录设备等。

也可以简单地理解为，计算机得到数据称为“输入”，计算机送出数据称为“输出”。可以形象地说，输入/输出就是人与计算机程序之间进行“对话”。此对话的过程是用户首先通过输入设备进行数据的输入，计算机程序接收到输入的数据后对数据进行一定的处理，将处理后的结果通过输出设备输出给用户，此过程如图 2.17 所示。

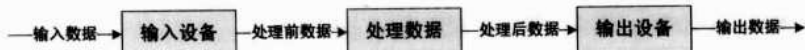


图2.17 输入/输出过程图

2.5.2 C 语言中输入/输出的实现

C 语言提供了许多常用函数（包括输入/输出函数、数学运算函数等），它们被存放在系统的函数库中。函数库就像是一个大仓库一样，存放了很多已经写好的函数，供使用者使用，但是这些函数不是 C 语言的组成部分。也就是说在 C 语言中，所有的数据输入/输出都是由库函数完成的，因此都是函数语句。例如，printf()函数（输出函数）和 scanf()函数（输入函数）。printf 和 scanf 是函数的名字，而不是 C 语言的关键字。

C 语言不提供输入/输出语句的目的是简化 C 语言编译系统，因为将语句翻译成二进制的指令是在编译阶段完成的，没有输入/输出语句就可以避免在编译阶段处理与硬件有关的问题，简化编译系统，而且通用性强，可移植性好，便于在各种计算机上实现。

由于 C 编译系统与 C 函数库是分别设计的，因此不同的计算机系统所提供的函数的数量、名字和功能不完全相同，这就给编程带来了很大不便。为了提高程序的通用性，在 C 语言的标准库“stdio.h”中，提供了一些通用的函数（如 printf()函数和 scanf()函数等），使得各种计算机系统都能调用，从而成为各种计算机系统的标准函数。而用来进行输入/输出的标准函数就叫做标准输入/输出函数。

在 C 语言中，常用的标准输入/输出函数有：printf()函数、scanf()函数、putchar()函数、getchar()函数、puts()函数和 gets()函数。其中，scanf()函数和 printf()函数叫做格式输入/输出函数；putchar()函数和 getchar()函数是字符输入/输出函数；gets()函数和 puts()函数是字符串输入/输出函数。

2.5.3 格式化输出——printf()函数

函数 printf()称为格式输出函数，其名称最末一个字母 f 即为“格式”（format）之意。其功能是把各种数据类型的数据（包括变量和常量）按照指定的格式转化为字符并在标准输出设备——显示器上显示出来。



1. printf()函数的格式

printf()函数是一个标准库函数，其函数原型包含在头文件“stdio.h”中。作为一个特例，不要求在使用 printf()函数之前必须包含“stdio.h”头文件，但是有些编译器要求必须包含头文件，所以建议在使用时包含头文件“stdio.h”。printf()函数调用的一般形式为：

```
printf("格式控制字符串", 输出表列);
```

其中，格式控制字符串用于指定输出格式，由格式字符串和非格式字符串两种组成。格式字符串是以%开头的字符串，在%后面跟有各种格式字符，以说明输出数据的类型、形式、长度、小数位数等。非格式字符串在输出时原样输出，在显示中起提示作用，所以也叫提示字符串。

输出表列中给出了各个输出项，即要输出的项目。其可以是变量，也可以是常量，甚至可以是表达式。要求格式字符串和各输出项在数量和类型上应该一一对应。

2. printf()函数格式字符串形式

```
%[*] [标志] [输出最小宽度] [.精度] [长度] 类型
```

其中方括号[]中的项为可选项。下面将对 printf()函数格式字符串的一般形式中的各部分进行说明

3. “类型”参数

“类型”参数用以表示输出数据的类型。如：输出整数时使用“%d”符号，输出字符时使用“%c”符号。这些符号被称为转换说明符，因为其指定了如何将数据转换成可以显示的形式。各转换说明符的意义如表 2.6 所示。

表2-6 转换说明符意义表

转换说明符	意义
%d	以十进制形式输出带符号整数（正数不输出符号）
%o	以八进制形式输出无符号整数（不输出前缀 0）
%x, %X	以十六进制形式输出无符号整数（不输出前缀 0x）
%u	以十进制形式输出无符号整数
%f	以小数形式输出单、双精度实数
%e, %E	以指数形式输出单、双精度实数
%g, %G	以%f或%e中较短的输出宽度输出单、双精度实数
%i	与%d相同
%c	输出单个字符
%s	输出字符串
%p	输出指针
%%	输出一个百分号

转换说明符使用方法程序清单如下所示：

```
#include "stdio.h"
void main()
{
```

```

int a=100;                //定义整型变量 a
float b=123.1234567;      //定义单精度浮点型变量 b
double c=12345678.1234567; //定义双精度浮点型变量 c
char d='p';              //定义字符变量 d
printf("a=%d, %o, %x, %u, %c\n",a,a,a,a,a); //以各种类型输出变量 a
printf("b=%f, %g, %e\n",b,b,b);           //以各种类型输出变量 b
printf("c=%f, %g, %e\n",c,c,c);           //以各种类型输出变量 c
printf("d=%c, %d\n",d,d);                 //输出变量 d
}

```

本程序在 Visual C++ 6.0 环境下运行结果如图 2.18 所示。

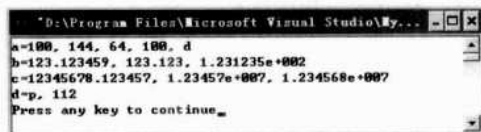


图2.18 输出结果图

注意：具体的输出结果视计算机的软、硬件环境不同可能会有所不同。

上面的程序中，第 8 行语句完成对被赋值为 100 的 int 型变量 a 按照各种形式输出。其处理过程是：

- ☐ “a=” 为提示字符串，所以原样输出。
- ☐ 将数据 a 按照转换说明符 “%d”、“%o”、“%x”、“%u” 和 “%c” 转换为 100 的十进制数、八进制数、十六进制数、无符号十进制数，以及字符形式。
- ☐ 将其以字符的形式输出在显示器屏幕上。

第 9 行和第 10 行语句分别完成了对赋值为 123.1234567 和 12345678.1234567 的 float 型数据 b 和 double 型数据 c 按照各种形式的输出。其处理过程是：首先原样输出提示字符串 “b=” 和 “c=”；然后按照转换说明符 “%f”、“%g”、“%e” 将 b 和 c 转换为 123.1234567 和 12345678.1234567 的小数形式、小数和指数形式中较短形式、指数形式并将其以字符的形式输出。

注意：输出值是对原来数据值进行了截断，截断的位数和方式与编译环境、计算机安装的操作系统甚至硬件都有关系。所以实际的输出结果有可能和图 2.18 有所不同。

第 11 行语句完成对被赋值为 p 的字符型变量 d 按照各种形式输出。其处理过程是：首先原样输出提示字符串 “d=”；然后按照转换说明符 “%c” 对字符原样输出，最后按照转换说明符 “%d” 将变量 d 转换为整数并以字符形式输出。

4. 转换说明符的使用方法

通常情况下，要使转换说明符与要输出的值的数据类型相匹配。当然，一般情况下是有多种选择的。例如，如果要输出一个 int 类型值，可以使用 %d、%x 或者 %o 的形式，其都能输出 int 类型的值，只不过表现形式有所不同而已。同样，对于 double 类型的值可以分别使用 %f、%g、%e 的形式输出。

如果转换说明符和输出的值的类型不匹配，会进行数据类型转换。输出的结果是经过类



型转换后的结果，但不一定是表示这个数据本来的值。例如，转换说明符使用方法的第 11 行用“%c”的形式输出了变量 d。

假设系统中 int 型数据占 2 个字节，char 型占 1 个字节，因为变量 a 被赋值为 100，而 100 小于 255（ASCII 码的最大值），字符“d”的 ASCII 码值为 100，所以输出了字符“d”。但是如果将 a 的值赋予 300，大于 255，这时，printf() 函数截断 2 个字节中的 1 个字节。这种截断相当于用 256 除这个数，并求余数。求出余数为 44，也就是字符“,” 的 ASCII 码值，所以会输出字符“,”。

所以，除非有特殊需要，例如想看看某个数字的 ASCII 码值是多少，一般情况下应该保证转换说明符与要输出的值的数据类型相匹配。

5. “标志”参数

下面来看 printf() 函数格式字符串的一般形式中的“标志”部分。标志参数表示输出数据的一些特征，例如对齐方式、正负号、前缀等，而不影响数据的值。如果希望输出的数据带有某种固定的特征或者格式，就应该根据需要加入各种标志。标志的意义如表 2.7 所示。

表2.7 标志意义表

标 志	意 义
-	结果左对齐，右边填充空格
+	输出符号（正号或负号）
空格	输出值为正时冠以空格，为负时冠以负号
#	对%c、%s、%d、%u 无影响；对%o 在输出时加前缀 0；对%x 在输出时加前缀 0x；对%e、%g、%f 当结果有小数时才给出小数点


一般形式中的“标志”主要是指 C 语言中用来实现诸如换行、清屏或 Tab 符号等功能等特殊标记，这些特殊标志意义如表 2.8 所示。

表2.8 特殊标志意义表

标 志	意 义
\n	换行
\f	清屏并换页
\r	回车
\t	Tab 符
\xhh	表示一个 ASCII 码用十六进制数表示，其中 hh 是 1 或 2 个十六进制数

各种特殊标志使用方法程序清单如下所示：

```
#include "stdio.h"
void main()
{
    int a=100;           //定义变量 a
    printf("a=%d\n",a);  //输出 a 和换行
    printf("a=%d\t",a);  //输出 a 和 Tab
    printf("a=%d\x23",a); //输出 a 和字符“#”
    printf("a=%d\f",a);  //输出 a 和清屏
}
```



具体的输出结果视计算机的软硬件环境可能会有所不同。上面的程序中，第 5 行语句中含有“\n”，所以在输出了变量 a 的值后输出了一个换行。第 6 行语句中含有“\t”，所以在输出了变量 a 的值后输出了一个 Tab 符。第 7 行语句中含有“\x23”，所以在输出了变量 a 的值后输出了一个“#”（ASCII 码用十六进制数表示为 23）。第 7 行语句中含有“\f”，所以在输出了变量 a 的值后进行了清屏并换到了下一页，如果还有输出的话将在下一页中显示。

printf()函数格式字符串的一般形式中的“输出最小宽度”是指用十进制整数来表示输出的最少位数。若实际位数多于定义的宽度，则按实际位数输出，若实际位数少于定义的宽度则补空格或0。指定足够大的固定字段宽度可以使输出更加整齐清晰。

```

D:\Program Files\Microsoft Visual Studio\My...
100 200 300
10000 20000 30000
1000000 2000000 3000000
Press any key to continue

```

如果在输出时，指定了足够大的字段宽度，输出将变得整齐清晰。例如将上面的语句改为：

那么输出结果如图 2.21 所示。

```


D:\Program Files\Microsoft Visual Studio\My...
100      200      300
10000    20000    30000
1000000  2000000  3000000
Press any key to continue

```

7. “精度”参数

函数 `printf()` 格式字符串的一般形式中的“精度”需以“.”开头，后跟十进制整数。其意义是：如果输出为数字，则表示小数的位数；如果输出的是字符，则表示输出字符的个数；若实际位数大于所定义的精度数，则截去超过的部分。



 **注意：**如果输出浮点数时指定的精度数小于实际位数时，会截去超过的部分进行输出，但是数据的值并没有变，如果输出后再次使用变量进行运算，变量的值还是运算最近的一次赋值。

8. “长度”参数

一般形式中的“长度”参数分为 h 和 l 两种，其中 h 表示按短整型量输出，l 表示按长整型量输出。需要说明的是，如果输出时用的长度和实际的数据类型不符，就无法保证输出结果的正确性。所以，建议在输出时选择和定义的数据类型匹配的长度。例如，如果定义了 short 型的变量，就以“%hd”的形式输出，如果定义了 long 型的变量，就以“%ld”的形式输出。

9. “*” 修饰符

通过前面的介绍可知，语句“printf(“%5d”,a);”将变量 a 以宽度为 5 的形式输出。宽度 5 是在程序运行之前就要确定好的。而使用“*”修饰符可以在程序运行的时候才确定此宽度，即可以实现动态宽度输出。

在使用“*”修饰符实现动态宽度输出时，如果输出语句中的格式包含“%*d”，那么参数表列中应该包括一个“*”的值和一个 d 的值。同理，“*”修饰符也可以与浮点数一起使用来指定动态的精度和宽度。动态宽度和精度的输出程序清单如下所示：

```
#include "stdio.h"
void main()
{
    int length;                //定义变量 length
    int precision;             //定义变量 precision
    int a=100;                 //定义变量 a 并赋值
    double b=123.456;         //定义变量 b 并赋值
    printf("请输入输出整型数据的宽度: ");
    scanf("%d",&length);
    printf("%*d\n",length,a);  //以设定宽度显示 a 的值
    printf("请输入输出浮点型数据的宽度和精度: ");
    scanf("%d%d",&length,&precision);
    printf("%*.*f\n",length,precision,b); //以设定宽度和精度显示 b 的值
}
```

上面的程序中，变量 length 指定了宽度，而 precision 指定了精度。第 9 行和第 12 行语句的作用是使用 scanf() 函数从键盘上输入值并赋给变量 length 和 precision，关于 scanf() 函数将在下一小节中详细介绍。

第 10 行语句的作用是以指定宽度 length 输出变量 a，因为 * 在 d 之前，所以 printf() 函数的参数表列中 length 在整型变量 a 之前。同样，第 13 行语句的作用是以指定宽度和精度输出变量 b，因为采用了“%*.*f”的形式，“.”前面的 * 表示宽度，后面的 * 表示精度，所以输出表列的顺序是宽度、精度和输出量。

2.5.4 格式化输入——scanf() 函数

函数 scanf() 称为格式输入函数，与 printf() 函数一样，名称的最后一个字母 f 表示“格式”(format)之意。其功能是从键盘上将用户输入的字符转换为用户预定类型的数据并赋值给

变量。

1. scanf()函数格式

函数 `scanf()` 是一个标准库函数，其函数原型在头文件 “`stdio.h`” 中，与 `printf()` 函数相同，C 语言也允许在使用 `scanf()` 函数之前不必包含 “`stdio.h`” 头文件。但是有些编译器要求必须包含头文件，所以建议在使用时包含头文件 “`stdio.h`”。`scanf()` 函数的一般形式为：

```
scanf("格式控制字符串", 地址表列);
```

其中，格式控制字符串的作用与 `printf()` 函数相同，但不能显示非格式字符串，也就是不能显示提示字符串。格式化字符串包括以下 3 类不同的字符。

- ☑ 格式化说明符：格式化说明符与 `printf()` 函数中的格式说明符基本相同。
- ☑ 空白字符：空白字符会使 `scanf()` 函数在读操作中略去输入中的一个或多个空白字符。
- ☑ 非空白字符：一个非空白字符会使 `scanf()` 函数在读入时剔除掉与这个非空白字符相同的字符。

地址表列是需要读入的所有变量的地址，而不是变量本身。这与 `printf()` 函数完全不同，要特别注意。各个变量的地址之间用 “,” 分开。地址是由地址运算符 “&” 后跟变量名组成的。例如：

```
&a, &b
```

分别表示变量 `a` 和变量 `b` 的地址。此地址就是编译系统在内存中给 `a`、`b` 变量分配的地址。

2. 使用&符号的规则

有关地址和指针的细节将在第 7 章中介绍，现在对于到底应不应该使用 & 符号，有两条简单的规则：

- ☑ 如果使用 `scanf()` 函数读取前面介绍过的基本类型变量的值，则要在变量名前面加入 &。
- ☑ 如果使用 `scanf()` 函数把一个字符串读进一个数组中，或者是给一个字符指针或者字符串赋值，不要使用 &。

说明 & 符号使用方法的程序清单如下：

```
#include "stdio.h"
void main()
{
    int i;                //定义整型变量 i
    float f;              //定义实型变量 f
    char c[20];           //定义字符串变量 c
    printf("请输入 int 型变量 i 和 float 型变量 f 的值: ");
    scanf("%d%f", &i, &f); //需要使用&
    printf("请输入字符串 c 的值: ");
    scanf("%s", c);       //不需要使用&
}
```

上面的程序中，第一个使用 `scanf()` 函数的语句中，因为变量 `i` 和 `f` 都是基本类型变量，`i` 和 `f` 表示变量本身，所以在用 `scanf()` 函数进行输入时，要加 & 符号表明是变量的地址。第二个使用 `scanf()` 函数的语句中，因为变量 `c` 是一个字符串，`c` 就表示一个地址，所以在用 `scanf()` 函数进行输入时，不需要加上 & 符号。



3. scanf()函数格式字符串的形式

`%[*][输入数据宽度][长度]类型`

其中有方括号“[]”的项为可选项。下面将对 `scanf()` 函数格式字符串的一般形式中的各部分进行说明，首先来看“类型”。类型参数用来表示输入数据的类型。函数 `scanf()` 使用的类型格式与函数 `printf()` 几乎完全相同。

主要的区别在于 `printf()` 函数把 `%f`、`%e`、`%E`、`%g` 和 `%G` 同时用于 `float` 和 `double` 类型，而 `scanf()` 函数只将其用于 `float` 类型，对于 `double` 类型，要在这些类型前面加字符“l”做修饰符。其转换说明符和意义如表 2.9 所示。

表2.9 转换说明符意义表

转换说明符	字 符 意 义
<code>%d</code>	输入十进制整数
<code>%i</code>	与 <code>%d</code> 作用相同
<code>%o</code>	输入八进制整数
<code>%x</code> 或 <code>%X</code>	输入十六进制整数
<code>%u</code>	输入无符号十进制整数
<code>%f</code> 或 <code>%e</code>	输入实型数（用小数形式或指数形式）
<code>%c</code>	输入单个字符
<code>%p</code>	输入一个指针（地址）
<code>%s</code>	输入字符串：输入的内容以第一个非空白字符作为开始，以下一个空白字符结束

宽度参数是指用十进制整数来指定的输入宽度，也就是输入字符的个数。下面的程序清单展示以不同的宽度输入数据：

```
#include "stdio.h"
void main()
{
    int b,c;                //定义变量的b和c
    scanf("%6d%4d",&b,&c);  //以固定宽度输入变量的值
    printf("b=%d\nc=%d\n", b,c); //输出变量的值
}
```

本程序在 Visual C++ 6.0 环境下运行结果如图 2.22 所示。

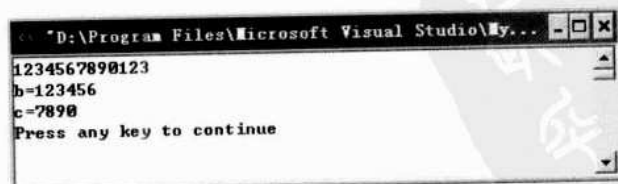


图2.22 输出结果图

上面的程序中，含有 `scanf()` 函数的语句先将输入的字符串截取前 6 个字符并转换为十进制整数赋值给变量 `b`，然后再截取从第 7 个字符开始的 4 个字符并转换成十进制整数赋值给变量 `c`。“长度”格式符为 `l` 和 `h` 两种，`l` 表示输入长整型数据（如 `%ld`）和双精度浮点数（如 `%lf`）。`h` 表示输入短整型数据。

4. scanf()函数读取输入的具体方式

- ☑ 使用一个%d 转换说明符来读取一个整数。

scanf()函数每次读取一个输入的字符,跳过空白字符(包括空格、制表符和换行符),直到碰到一个非空白符。因为要读取一个整数,所以scanf()函数期望发现一个数字字符或者一个符号(+或者-)。

如果发现了一个数字或者符号,就保存并读取下一个字符;如果接下来的字符仍然是一个数字,保存这个数字并继续读取下一个字符。持续读取和保存字符直到碰到一个非数字字符。这时,认为已经读取到整数的尾部,scanf()函数将这个非数字字符放回输入。所以在程序下一次开始读取输入时,将从前面被放弃的那个非数字字符开始。

- ☑ scanf()函数计算其读取到所有数字的相应数值,并将这个数值放入指定的变量中。

如果使用了字段宽度,scanf()函数会在字段结尾或者第一个非空白字符处(以最先到达的那个为准)结尾。使用其他的数字说明符读取输入与使用%d大致相同。主要区别在于scanf()函数会把更多的字符看成数字的一部分。例如%x说明符要求scanf()函数识别十六进制数字,也就是说,除了从0~9外,还有a~f和A~F。如果使用的是%c说明符,所有的输入字符都被认为是有效的字符。

如果下一个输入的字符是一个空白或者换行符,不会跳过它们,而是将其赋值给变量。如果使用的是%s说明符,那么空白字符以外的所有字符都被认为是有效字符。所以scanf()函数跳过空白字符直到第一个非空白字符,然后保存再次碰到空白字符之前的所有非空白字符。

假定输入了“hello,123”,那么保存的是“hello,123”。如果使用了字段宽度,scanf()函数在字段结尾处或者第一个空白字符处结束。当scanf()函数把字符串放进一个指定的字符数组时,将自动增加终止符“\0”,使得数组内容成为一个C语言的字符串。

5. 使用scanf()函数的注意事项

- ☑ scanf()函数中没有精度控制,如语句scanf("%6.3f",&a);是非法的。不能企图用此语句输入小数位数为3位的实数。
- ☑ scanf()函数中要求给出变量地址,如果给出变量名则会出错。如语句scanf("%d",a);(a为基本类型)是非法的,应改为scanf("%d",&a);。
- ☑ 在输入多个数值数据时,若格式控制串中没有非格式字符作为输入数据之间的间隔则可用空格、Tab或回车作间隔。C编译在碰到空格、Tab、回车或非法数据(如对“%d”输入“12B”时,B即为非法数据)时即认为该数据结束。
- ☑ 在输入字符数据时,若格式控制串中无非格式字符,则认为所有输入的字符均为有效字符。如语句scanf("%c%c",&a,&b);,当输入为AB时,函数把字符'A'和字符'B'中间的字符' '也当做有效字符,所以将输入的第一个字符'A'赋值给变量a,将第二个字符' '赋值给了变量b,而不是把字符'B'赋值给变量b。
- ☑ 如果格式控制串中有非格式字符则输入时也要输入该非格式字符。如语句scanf("a=%d,b=%d",&a,&b);,字符'a','=',' ',以及'b'都属于非格式字符,在输入时,应该原样并按照位置输入。例如输入为“a=123,b=456”,函数会把123赋值给变量a,把456赋值给变量b。但是如果输入为“123,456”的话,始终没有输入非格式字符,



所以变量 a 和变量 b 都不会被赋值，系统给其一个默认值，而这个默认值往往不是预期的值。

- ❑ 如输入的数据与输出的类型不一致时，虽然编译能够通过，但结果将不正确。如定义一个 int 型数据 a 并赋值为 123456，以 scanf("%d",&a); 作为输入语句，输入数据类型为整型，而如果以 printf("%hd",a); 作为输出语句，输出语句的格式串中说明为短整型，因为 a 的值超出了短整型的范围，所以输出的结果将不会是 123456，在笔者的系统和编译环境下结果为 -7616，具体的输出结果会根据系统和编译器的不同而不同。
- ❑ 如果出现 "%d\n" 的形式，在输入时要多输入一行函数才返回。这是因为 "\n" 在 scanf() 函数格式串中不表示等待换行符，而是读取并放弃所有的空白字符。因此，"%d\n" 中的 "\n" 会使 scanf() 函数读到非空白字符为止，而 scanf() 函数需要读到下一行才能找到这个非空白字符，所以程序多输入一行函数才返回。
- ❑ scanf() 函数中的 "*" 修饰符与 printf() 函数中的 "*" 修饰符提供了截然不同的作用。当将其放在 % 和格式字母之间时，其用以表示该输入项，读入后不赋予相应的变量，即跳过该输入值。

下面的程序清单展示 scanf() 函数中 "*" 的用法：

```
#include "stdio.h"
void main()
{
    int a,b;                //定义变量的 a 和 b
    scanf("%d %*d %d",&a,&b); //使用包含*的形式输入变量的值
    printf("%d,%d",a,b);    //输出变量的值
}
```

本程序在 Visual C++ 6.0 环境下运行结果如图 2.23 所示。

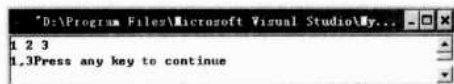


图2.23 输出结果图

上面的程序中，含有 scanf() 函数的语句对输入的字符“1 2 3”进行了这样的处理：首先根据“%d”把输入的数据 1 转换成十进制整数并赋值给变量 a，而后因为“%*d”中含有“*”修饰符，所以输入的字符 2 被跳过，最后根据“%d”把输入的数据 3 转换成十进制整数并赋值给变量 c。

2.5.5 字符的输出——putchar()函数

putchar() 函数的功能是向标准输出设备（显示器）输出一个字符，其函数原型包含在头文件 stdio.h 中，在使用前必须包含头文件 stdio.h。其调用格式为：

```
putchar(ch);
```

其中 ch 为一个字符变量或常量。putchar() 函数的作用等同于 printf("%c", ch);，此外，使用 putchar() 函数应注意两个问题：首先，putchar() 函数只能输出单个字符；其次使用本函数前必须包含文件 stdio.h，否则编译器会报错。

2.5.6 字符的输入——getchar()函数

1. getchar()函数格式

getchar()函数的功能是从标准输入设备(键盘)上输入一个字符,其函数原型包含在头文件 stdio.h 中,在使用前必须包含头文件 stdio.h。其调用格式为:

```
getchar();
```

通常把输入的字符赋予一个字符变量,构成赋值语句,如:

```
char c;  
c=getchar();
```

要从键盘上输入一个字符并将其输出在屏幕上,程序清单如下所示:

```
#include "stdio.h"  
void main()  
{  
    char c;  
    printf("input a character:\n");  
    c=getchar();        //从键盘上输入一个字符  
    putchar(c);         //输出输入的字符  
}
```

上面的程序中,第6行语句的作用是使用 getchar()函数从键盘上获取用户输入的单个字符并赋值给变量 c。第7行语句的作用是用 putchar()函数将变量 c 的值显示在显示器上。

2. 使用 getchar()函数应注意的问题

- ☑ getchar()函数只能接收单个字符,输入多于一个字符时,只接收第一个字符。
- ☑ 在 getchar()函数中输入数字也按字符处理。例如,在语句 c=getchar();中输入 1,那么其实输入的是字符'1',而非数字 1。如果用格式输出语句 printf("%d",c);输出 c 的值的的话,输出的结果是 c 的 ASCII 码。
- ☑ 使用本函数前必须包含文件 stdio.h;否则,编译器会报错,无法编译通过。
- ☑ 在 Visual C++ 6.0 屏幕下运行到本函数时,将退出 Visual C++ 6.0 屏幕进入用户屏幕等待用户输入。输入完毕再返回 Visual C++ 6.0 屏幕。

2.5.7 输出字符串——puts()函数

1. puts()函数格式

puts()函数是用于向标准输出设备(显示器)输出字符串并换行的函数,其函数原型包含在头文件 stdio.h 中,在使用前必须包含头文件 stdio.h。其调用格式为:

```
puts(s);
```

其中 s 为字符串变量(字符串数组名或字符串指针)。也可以直接将字符串作为参数进行输出。

2. .puts()函数格式实例——输入字符串

可以用下面的语句输入字符串“Hello,World”:

```
puts("Hello,World");
```



puts()函数的作用与 printf("%s\n", s)相同。需要说明的是, puts()函数只能输出字符串,不能输出数值或进行格式变换。例如,想通过语句“puts(a);”输出变量 a,而变量 a 又被定义为 int 类型的话,那么由于 puts()函数不能将 a 转换为字符型数据并输出,结果将导致程序产生编译出错。

2.5.8 读取字符串——gets()函数

1. gets()函数格式

gets()函数用来从标准输入设备(键盘)读取字符串直到回车符结束,但回车符不属于这个字符串。其函数原型包含在头文件 stdio.h 中,在使用前必须包含头文件 stdio.h。其调用格式为:

```
gets(s);
```

其中 s 为字符串变量(字符串数组名或字符串指针)。

2. gets(s)函数与 scanf("%s", &s)

gets(s)函数与 scanf("%s", &s)相似,但不完全相同。使用 scanf("%s", &s)时,函数输入字符串时存在一个问题,就是如果输入了空格会认为输入字符串结束,空格后的字符将作为下一个输入项处理,但 gets()函数将接收输入的整个字符串直到回车为止。

3. gets(s)函数实例——输出姓名和姓

下面的程序完成一个有趣的功能:要求你两次输入你的姓名,在第一次输入后程序输出你的姓名,在第二次输入后程序输出你的 first name,对于以第一个字为姓氏的人来说,就是输出这个人的姓。输出姓名和姓的程序清单如下:

```
#include "stdio.h"
void main()
{
    char name[20];
    printf("Please input your name\n");
    gets(name);          //用 gets()函数输入姓名
    puts("Your name is:");
    puts(name);           //输出姓名
    puts("Please input your name again");
    scanf("%s", &name); //用 scanf ()函数输入姓名
    puts("Your first name is:");
    puts(name);           //输出姓
}
```

本程序在 Visual C++ 6.0 环境下运行结果如图 2.24 所示。

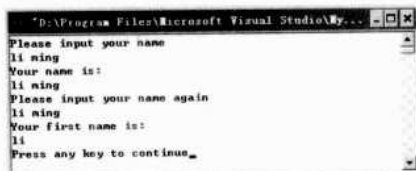



图 2.24 输出结果图

上面的程序中,第4行语句首先定义了一个长度为20的字符数组 `name` 用于存放输入的姓名。第6行语句使用 `gets()`函数接收用户从键盘上输入的字符串并赋值给 `name`,因为 `gets()`函数直到遇到回车键才认为输入完成,在键盘上输入“li ming”,按回车键,给 `name` 数组赋值为“li ming”,所以第8行语句使用 `puts()`函数输出了数组 `name` 的值“li ming”。而第10行语句使用 `scanf()`函数接收用户输入并赋值给 `name`,因为 `scanf()`函数遇到空格键就认为输入完成,按照上面的输入,给 `name` 数组赋值为“li”,所以第12行语句使用 `puts()`函数输出了数组 `name` 的值“li”。

注意: `gets(s)`函数中的变量 `s` 为一个字符串。如果是单个字符,编译连接不会有错误,但运行后会出现“Null pointer assignmemt”的错误。



第 3 章

运算符、表达式和语句



C 程序是一种模块化的程序，其模块就是函数，即一个 C 程序是由若干个函数组成的。一个函数的组成部分除了第 2 章中介绍的变量定义外，另一个重要的组成部分就是语句。许多语句是用来进行运算的，一般形式是表达式加分号构成的。而表达式由常量和变量再加上运算符构成。本章就从运算符入手，来介绍 C 语言中的运算符、表达式和语句。此外，需要说明的是，在本章中所有例子均假设系统中 short 型数据占 2 个字节，int 型数据占 2 个字节，unsigned int 型数据占 2 个字节，long 型数据占 4 个字节，unsigned long 型数据占 4 个字节，float 型数据占 4 个字节，double 型数据占 8 个字节。



3.1 运算符和表达式概述

运算是所有程序均要执行的最基本的操作。而运算是由运算符和常量（或变量）组成的表达式来实现的。

3.1.1 运算符

运算符是用来表示各种不同运算的符号，其作用是告诉计算机执行某些数学或者逻辑操作。运算是针对数据进行加工的过程。参加运算的数据称为运算对象、运算量或操作数。运算对象可以是一个、两个或三个，对应地将运算符称为单目运算符、双目运算符和三目运算符。最常用的是双目运算符，例如，“=”就是一个双目运算符。

C 语言含有丰富的运算符，共有 30 余种，大致可分为以下几类：

- ☑ 算术运算符（+、-、*、/、%）。
- ☑ 自增自减运算符（++、--）。
- ☑ 赋值运算符，包括简单赋值运算符（=）和复合赋值运算符（+=、-=、*=、/=等）。
- ☑ 强制类型转换运算符[(类型符)]。
- ☑ 关系运算符（<、<=、>、>=、==、!=）。
- ☑ 逻辑运算符（!、&&、||）。
- ☑ 位运算符，包括移位运算符（<<、>>）和按位运算符（~、&、|、^）。
- ☑ 求字节数运算符（sizeof）。
- ☑ 逗号运算符（,）。
- ☑ 条件运算符（?:）。
- ☑ 指针运算符（*、&）。
- ☑ 下标运算符（[]）。
- ☑ 分量运算符（.、->）。

C 语言中的运算符都是键盘上的单个符号（如+、-、*、/等）或若干个符号的组合（如++、--、+=、*=等）。有些运算符有双重含义，如运算符“+”既表示单目的取正运算，又表示双目的加法运算；运算符“*”既表示单目的指针运算（取变量运算），又表示双目的乘法运算；运算符“&”既表示单目的指针运算，又表示双目的按位与运算。

3.1.2 表达式

强调表达式是 C 语言的一个显著特征。表达式是显示如何运算的公式，其目的是获得值。最简单的形式是变量和常量。变量表示程序运行时运算出的值；常量表示不变的值。更为复杂的表达式把运算符用于运算对象。例如，下面的表达式：

```
x-(5*y)
```

其将运算符“-”用于变量 x 和(5*y)，而(5*y)也是一个表达式，即表达式的运算对象又可以是表达式，将其称为原表达式的子表达式。



每个表达式都有一个值，就是其运算后的结果。表达式必须有一个特定类型，即结果的数据类型。比如下面的表达式的类型为 `int`：

```
5+10
```

如果表达式没有产生值，类型就为 `void`。

表达式可以按照使用的运算符进行分类，比如，下面的表达式：

```
x+y
```

因为该表达式使用了算术运算符“+”，所以将其称为算术表达式。

3.1.3 运算符的优先级和结合性

当各种不同的运算符组成表达式时，运算符的优先级和结合性将起到十分重要的作用。对每个运算符，除了要把握其功能、对运算对象的类型要求外，还要把握运算符的优先级和结合性。

1. 隐含二义性

当表达式包含多个运算符时，在解释此表达式时就产生了困难。例如，表达式 `x+y*z`，是该理解为 `x` 加上 `y` 的结果再乘以 `z` 呢？还是该理解为 `y` 乘以 `z` 的结果再与 `x` 相加呢？这就产生了隐含二义性，即包含了两种都可能正确的意思。

2. 运算符优先级的规则

C 语言通过采用运算符优先级的规则来解决这种隐含二义性，运算符的优先级共分为 15 级，其中 1 级最高，15 级最低。在表达式中，优先级较高的先于优先级较低的进行运算，例如：

```
x+y*z    //先求 y*z 的值，然后再与 x 相加
-x*-y     //等于 -x 的值乘以 -y 的值
```

3. 运算符的结合性

当一个运算对象两侧的运算符优先级相同时（如 `x*y%z`），按运算符的结合性所规定的结合方向处理。C 语言中各运算符的结合性分为以下两种：

- ☑ 左结合（自左至右结合）。
- ☑ 右结合（自右至左结合）。

二元算术运算符就是左结合的。例如，表达式 `x-y+z`，运算时 `y` 应先与“-”号结合，执行 `x-y` 运算，然后其结果再执行 `+z` 的运算。最典型的右结合的运算符是赋值运算符。例如，表达式 `x=y=z`，由于“=”是右结合，应先执行 `y=z` 再执行 `x=(y=z)` 运算。

4. 使用圆括号（`()`）分组运算符

C 语言含有丰富的运算符，此处却成为记忆运算符优先级和结合性的一个障碍，使得使用运算符时常因此而出错。其实，有一个简单有效的办法可以解决此问题，那就是在表达式中使用圆括号（`()`）进行分组。例如：


```
x+(y*z)
```

先进行 `y*z` 运算然后将其结果再与 `x` 相加。又例如：

```
(x+y)*z
```

先进行 `x+y` 运算然后将其结果再与 `z` 相乘。



 **注意：**在表达式中使用圆括号（`()`）进行分组是一种很好的编程方式，它大大提高了程序的易读性。

3.2 算术运算符与算术表达式


算术运算符和算术表达式是编程语言中广泛使用的一种运算符和表达式。

3.2.1 算术运算符

算术运算符就是进行算术运算的运算符，可以执行加法、减法、乘法、除法和求余运算。C 语言的算术运算符如表 3.1 所示。

表3.1 算术运算符

运 算 符	意 义	范 例	结 果
<code>+</code> （一元）	正号	<code>+x</code>	<code>x</code> 的值
<code>-</code> （一元）	负号	<code>-x</code>	<code>x</code> 的算术负数
<code>+</code>	加法	<code>x+y</code>	<code>x</code> 和 <code>y</code> 的和
<code>-</code>	减法	<code>x-y</code>	<code>x</code> 和 <code>y</code> 的差
<code>*</code>	乘法	<code>x*y</code>	<code>x</code> 和 <code>y</code> 的积
<code>/</code>	除法	<code>x/y</code>	<code>x</code> 和 <code>y</code> 的商
<code>%</code>	求余	<code>x%y</code>	<code>x</code> 除以 <code>y</code> 的余数

 **注意：**一元运算符“`+`”无任何操作，主要是为了强调某数值常量为正。一般不使用此运算符。

除了运算符“`%`”外，表 3.1 中的二元运算符既允许运算对象是整数，也允许其是浮点数，甚至是两者的混合。

运算符“`/`”和“`%`”与数学中的除法和求余数有一些差别，在使用时要特别注意以下几点：

- ☑ 运算符“`/`”可能产生和预期不同的结果。当两个运算对象都是整数时，运算符“`/`”通过舍弃小数部分的方法截取结果。例如，`1/2` 的结果不是 0.5，而是 0。
- ☑ 运算符“`%`”要求两个运算对象都是整数，如果其中有一个不是整数，程序将编译出错。例如，`1%1.5` 是不能编译通过的。
- ☑ 当运算符“`/`”和运算符“`%`”的运算对象是负数时，其结果与具体实现有关。

如果两个运算对象中有一个是负数，那么除法的结果既可以向上取整，也可以向下取整。如果 `x` 或 `y` 是负数，`x%y` 的符号和具体实现有关。下面的程序清单说明了这些：

```
#include "stdio.h"

void main()
{
    int x,y;           //定义变量 x 和 y
    x=-12;             // x 赋值为负数
    y=5;               // y 赋值为正数
}
```



```
printf("相除的结果是: %d\n 求余的结果是: %d\n", x/y, x%y); //输出 x 和 y 的值  
}
```

本程序在 Visual C++ 6.0 环境下运行结果如图 3.1 所示。

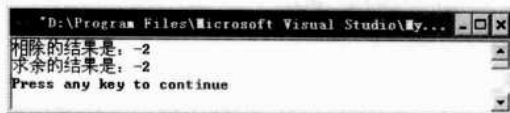


图3.1 运行结果图

程序的运行结果和具体的实现有关, 当运行的操作系统和编译器不同时, 结果将有所不同。就此程序而言, 相除结果可能会向下取整得-3, 求余的结果的符号也可能取正号使其变成 2。

3.2.2 算术表达式

用算术运算符和圆括号将运算对象连接起来的符合 C 语言语法规则的式子称为算术表达式。运算对象包括常量、变量、函数等。例如, 下面是一个合法的算术表达式:

```
3.14*pow(r,2) //等价于数学公式  $3.14 \times r^2$ 
```

其中, 包含运算对象 3.14 (常数) 和运算对象 pow(r,2), pow() 是一个 C 语言标准库函数, 此处的作用是求 r 的 2 次方。而下面是两个不合法的算术表达式:

```
7.5%3 //浮点数不能作为运算符%的运算对象  
2x+3y //不能省略运算符*
```

3.2.3 算术表达式的求值

如果没有圆括号, 算术表达式将按运算符的优先级进行求值。具体来说, 两个单目运算符的优先级最高:

第 1 步 对其进行遍历。

第 2 步 遍历的是运算符 “*”、“/”、“%”。

第 3 步 遍历两个双目运算符 “+” 和 “-”。

规则是单目运算符的优先级要高于双目运算符, 例如:

```
-a+b/2-c*5
```

假设 a 的值为 10, b 的值为 20, c 的值为 5, 则此表达式的运算过程如下: $-10+20/2-5*5 = -10+10-5*5 = -10+10-25 = 0-25 = -25$, 此过程如图 3.2 所示。

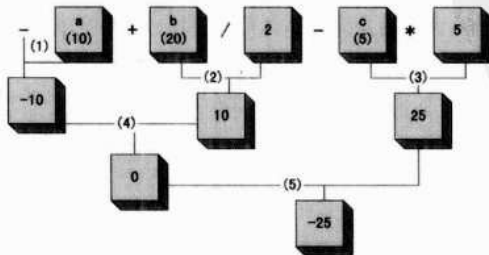


图3.2 表达式求值

此处要特别说明一点，标准 C 中对于很多运算符（包括算术运算符）的求值顺序并没有明确规定，实际顺序与具体实现有关。也就是说，上述步骤中的第二步和第三步并不固定，子表达式 $b/2$ 和 $c*5$ 哪个先执行取决于具体实现。

当然，这两个子表达式哪个先执行都不会影响表达式的值，但是有些情况下子表达式的求值顺序可能会对表达式的值产生影响。为了明确子表达式求值的顺序，可以使用圆括号来显式地确定此顺序。例如，上面的表达式可以写成下面的形式：

```
(-a+b/2)-c*5
```

此表达式会严格按照上述步骤来求值。

3.3 赋值运算符与赋值表达式

在第 2 章介绍变量赋值时，已经对赋值运算符进行了简单介绍。其实，前面介绍过的赋值运算符叫做简单赋值运算符，C 语言还有另外一种赋值运算符，叫做复合赋值运算符。本节就详细介绍这两种赋值运算符及赋值表达式。

3.3.1 简单赋值

在 C 语言中，“=” 是一个运算符，称为赋值运算符。由赋值运算符将一个变量和一个表达式连起来的式子称为赋值表达式。因其只是简单地给变量赋值，故也称其为简单赋值表达式。其一般形式为：

```
变量名=表达式
```

其中，“=” 叫做赋值号，其左边必须是一个代表某一个存储单元的变量名，其右边是 C 语言中合法的表达式（可以是任意类型的表达式）。

赋值运算的确切含义是把赋值号右边表达式的值存入以左边变量为标识的存储单元中去。例如， x 和 y 都是 `int` 类型变量：

```
x=100    //把常量 100 赋给变量 x
y=x+1    //把表达式 x+1 中的值赋给变量 y
```

变量的值是可以改变的，所以在程序中可以多次给一个变量赋值，变量的值就不断地被刷新，内存中变量的值就是最后一次的赋值。例如：

```
int x;    //声明整型变量 x
x=100;    //给变量 x 赋值 100
x=200;    //给变量 x 赋值 200
```

变量 x 中最终存储的值是 200。赋值运算符可以串联在一起，例如：

```
x=y=z=100
```

因为赋值运算符是右结合的，所以上面的表达式等价于：

```
x=(y=(z=100))
```

其运算过程是先将 100 赋给 z ，然后将子表达式 $z=100$ 的值赋给 y ，再将子表达式 $y=(z=100)$ 的值赋给 x 。最后 3 个变量的值都为 100。

在使用简单赋值时，需注意以下几点：



- ☑ 赋值运算符不同于数学中的“等于”符号，在 C 语言中，“=”只是赋值的意思，不是判断是否相等的意思。
- ☑ 表达式 $n=n+1$ 是合法的表达式，意思是取出变量 n 中原来的值加 1 后在放到变量 n 的存储单元中， n 的值被刷新为加 1 后的值。
- ☑ 赋值运算符的优先级只高于逗号运算符，比任何其他运算符的优先级都低，并且自右向左结合。

3.3.2 左值和右值

每个赋值表达式都必须有一个左值和一个右值。左值位于赋值运算符的左侧，与左值相对的右值则位于赋值运算符的右侧。

1. 左值

左值是指可以被赋值的表达式，表示存储在计算机内存中的对象。在赋值表达式中，左值必须是内存中一个可存储的变量。例如：

```
int x;
float f;
x=1;           //x 是此赋值表达式的左值
f=3.14;        //f 是此赋值表达式的左值
```

其中，变量 x 是一个整数，它对应于内存中的一个可存储位置，因此，在表达式“ $x=1$ ”中， x 就是一个左值。同样，变量 f 也是一个左值。赋值运算符的左值不能是常量或者表达式。例如：

```
#define CONST_VAL 10
int x
l=x;           //不能用整型常量作为左值
CONST_VAL = 100; //不能用符号常量作为左值
x+1=100;       //不能用表达式作为左值
```

在上述 3 条赋值语句中，无论是整型常量 1，符号常量 `CONST_VAL`，还是表达式 $x+1$ ，其值都不能改变，都不是左值。因此，这 3 条赋值语句中没有左值，无法编译通过。

2. 右值

右值可以被定义为能赋值的表达式，它出现在赋值运算符的右边。与左值不同，右值既可以是变量，也可以是常量或表达式。例如：

```
int x, y;
x = 1;           //右值为常量
y=(x+1);         //右值为表达式
x=y;            //右值为变量
```

一个赋值表达式必须有一个左值和一个右值。因此，下述程序段无法编译通过，因为该赋值表达式缺少一个右值。

```
int x;
x=void_function(); //右值为一个无返回值的函数
```

因为函数 `void_function()` 是无返回值的，即赋值号右边没有值。赋值表达式没有右值，所以会编译出错。



3.3.3 复合赋值

在编程时,经常需要利用变量原有的值计算出新值并重新赋值给此变量。例如,下面的表达式就是把变量 i 加 2 后重新赋值给自己:

```
i=i+2
```

C 语言中的复合赋值运算符允许缩短此类表达式,可以使用 $+=$ 运算符将上面的表达式简写为:

```
i+=2
```

C 语言可以使用 10 种复合赋值运算符,其中与算术运算符相结合的复合赋值运算符如表 3.2 所示。

表3.2 部分复合赋值运算符

运 算 符	范 例	结 果
$+=$	$i+=2$	$i=i+2$
$-=$	$i-=2$	$i=i-2$
$*=$	$i*=2$	$i=i*2$
$/=$	$i/=2$	$i=i/2$
$\%=$	$i\%=2$	$i=i\%2$

复合赋值运算符与简单赋值运算符的优先级相同。其一般形式为:

变量 双目运算符=表达式

其等价于:

变量=变量 运算符 表达式

例如:

```
n+=10 //等价于 n=n+10
```

```
x*=2+3 //等价于 x=x*(2+3)
```

其结合性是自右向左结合,例如,下面的表达式:

```
x+=x-=1
```

等价于:

```
x+=(x-=1)
```

3.3.4 赋值运算符的副作用

在编程时,通常不希望运算符修改其运算对象,因为数学上的运算符就是如此。例如 $x+y$ 会求出两个数的和,但是不会改变 x 或 y 的值。

C 语言中的大多数运算符(如算术运算符)不会改变运算对象的值,但也有一些运算符会改变此值。因为这种运算符进行的已不仅仅是计算值,所以称其有副作用。赋值运算符就是一种具有副作用的运算符。例如,下面的赋值表达式均改变了运算对象的值:

```
i=8*100
```

```
j=i/20
```

```
k+=j
```




其中，第一个表达式产生了副作用，将 800 赋值给了 i。同样，第二个和第三个表达式也改变了变量 j 和 k 的值，产生了副作用。

3.3.5 子表达式的求值顺序

前面已经介绍过，表达式中的子表达式的求值顺序会依据具体实现而不同。并且给出了一种解决方案，就是使用圆括号“()”。但是，某些情况下，即使使用圆括号()也不能确定表达式的求值顺序，特别是包含具有副作用的赋值运算符的表达式。例如：

```
y = (x+1) - (x=2)
```

假设 x 的初始值为 10，则此表达式的值有可能是 9，也有可能是 1。两种结果的运算过程如图 3.3 所示。

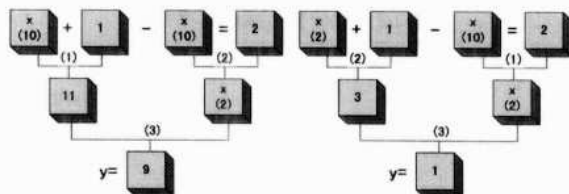


图3.3 求值顺序图

为了避免上述问题，在此提出了一个好的解决方案：不在子表达式中使用赋值运算符，而是采用一连串分离的赋值表达式。例如，如果能让上面的表达式结果为 9，则可以分解成下面几个表达式：

```
z = x + 1
x = 2
y = z - x
```

而要使表达式的值为 1，则可以分解成下面几个表达式：

```
x = 2
y = x + 1 - x
```

3.4 类型转换

当表达式的运算对象类型不一致时，就会发生类型转换。C 语言的类型转换，可以分为两种：自动类型转换（隐式类型转换）和强制类型转换（显示类型转换）。本节就来详细介绍这两种类型转换。

3.4.1 类型转换简述


计算机在执行算术运算时，具有很多的限制。一般情况下，计算机在执行算术运算时，运算对象具有相同的大小（具有相同数量的位），并且要求具有相同的存储方式。比如，计算机可以直接将两个 16 位整数相加，也可以将两个 32 位的浮点数相乘。但是不能直接将 16 位整数和 32 位整数相加，也不能直接将 32 位整数和 32 位浮点数相乘。

另一方面，C 语言允许在表达式中混合使用基本类型数据。一个表达式中可以组合使用整数、浮点数，甚至字符。为了使硬件能对其进行运算，C 语言编译器一般会生成一些指令将某

些运算对象转换成某种类型，称之为类型转换。例如，下面对于 16 位 int 型变量 a 和 32 位 long int 型变量 b 相加，编译器将会将 a 的类型转换为 32 位的值，一般也是 long int 型。

`a+b` //表达式的值是 long int 型

在类型转换中，有时会将“比较窄的”（指占用的位少）运算对象转换为“比较宽”（指占用的位多）运算对象，不会丢失信息。例如，将 16 位的 int 型转换成 32 位的 long 型。而此种转换相当于提高了运算对象的宽度，所以称其为提升。而有时会将“比较宽的”运算对象转换为“比较窄的”运算对象，则可能会发生数据丢失。例如，将 1.68 转换为整数将变为 1，相当于对原值进行了截取，故将其称为截断。


 **注意：**如果程序中可能产生截断，大多数编译器会提出警告，但程序仍能编译通过。由于截断会产生数据丢失，所以要特别小心使用。

根据能否自动处理此种转换，可以将类型转换分为自动类型转换和强制类型转换。如果编译器自动处理转换而不需要程序员介入，叫做自动类型转换或者隐式类型转换。而程序员通过强制类型转换运算符进行类型转换，叫做强制类型转换或者显示类型转换。下面分别进行介绍。

3.4.2 自动类型转换

在 C 语言中，在下列情况下会发生自动类型转换：

- ☑ 当算术表达式或逻辑表达式中操作数的类型不相同时。此种转换一般称为常用算术转换。
- ☑ 当赋值运算符右侧表达式的类型和左侧变量的类型不匹配时。
- ☑ 函数调用中的实参和与其对应的形参不匹配时。
- ☑ 当 return 语句中表达式类型和函数返回值类型不匹配时。

 **提示：**本节只介绍前两种情况，后两种情况将在第 5 章“函数”中详细介绍。

1. 常用算术转换

常用算术转换多用于二元运算符的运算对象上，包括算术运算符、关系运算符。其一般都是提升的，不会产生数据丢失，即是安全的。其转换规则如图 3.4 所示。

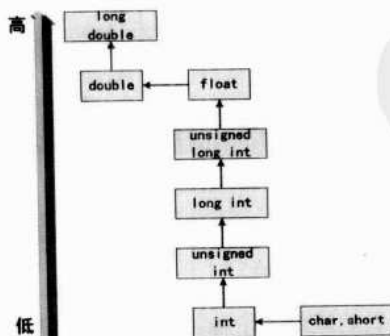


图3.4 类型转换规则图



上图中横向箭头表示必须得转换，其将箭头右面的类型转换为箭头左面的类型。如两个 float 型数参加运算，虽然它们的类型相同，但仍要先转换成 double 型再进行运算，运算结果也为 double 型。纵向箭头表示当运算符两边的运算数为不同类型时，进行提升转换，即将箭头下面的类型转换成箭头上面的类型。如一个 long 型数据与一个 int 型数据一起运算，需要先将 int 型数据转换为 long 型，然后两者再进行运算，运算结果也为 long 型。此处用一个程序说明此规则，程序清单如下所示：

```
void main()
{
    /*定义各种类型变量*/
    char c='a';
    short s=100;
    int i=200;
    unsigned int u=60000;
    long l=100000;
    float f=1.23456E6;
    double d=1.234567E16;
    s+c;    //表达式类型为 int
    i+c;    //表达式类型为 int
    u+i;    //表达式类型为 unsigned int
    l+i;    //表达式类型为 long
    f+l;    //表达式类型为 float
    d+f;    //表达式类型为 double
    f+f;    //表达式类型为 double
}
```

上面程序中的表达式的值在内存中的表示如图 3.5 所示。

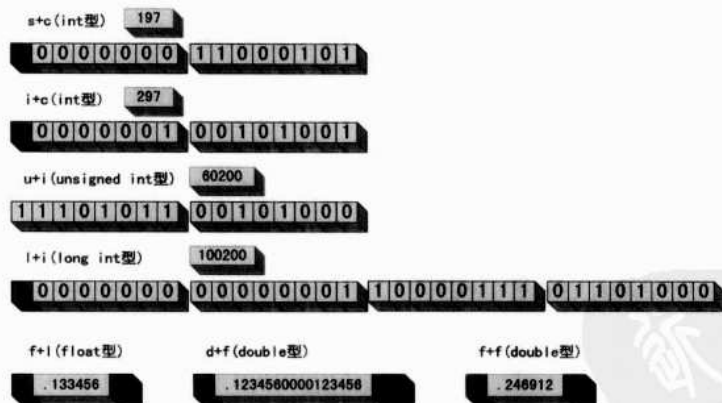


图3.5 表达式在内存中的表示

注意：为了表示方便，图中的 float 型和 double 型数据是用十进制数表示的，但其在计算机中是用二进制数表示的。

从图 3.5 中可以看出，当表达式的两个运算对象分别是有符号数和无符号数时，会通过将表示符号的位看作表示数的位的方法将有符号数转换成无符号数。此规则可能会引起一些

隐蔽的编程错误。

例如，假设上面程序中变量 *i* 的值为 -10，变量 *u* 的值为 10，但表达式 *u+i* 的值却并不是 0。这是因为变量 *i* 首先转换成 `unsigned int` 型，而负数不能被表示为有符号数，所以转换后其数值不再是 -10，而会变成一个很大的正数，用此数加上 10，其结果当然不是 0。

所以，要尽量避免使用无符号整数，特别是不要和有符号整数混合使用。

2. 赋值中的转换

当赋值运算符两边的运算对象类型不同时，会发生类型转换。转换的规则是：把赋值运算符右侧表达式的类型转换为左侧变量的类型。此转换有时是提升的，有时是截断的。各种类型的具体转换规则如下。

☐ 浮点型与整型


- 将浮点数（单、双精度）转换为整数时，将舍弃浮点数的小数部分，只保留整数部分。
- 将整型值赋给浮点型变量，数值不变，只将形式改为浮点形式，即小数点后带若干个 0（0 的个数与具体实现有关）。例如：

```
int i=100.123; //i 的值将为 100
float f=100; //f 的值将为 100.000000
i=123.123-i; //123.123-i 的值为 23.123, i 的值为 23
```

☐ 单、双精度浮点型

由于 C 语言中的浮点值总是用双精度表示的，所以 `float` 型数据只是在尾部加 0 延长为 `double` 型数据参加运算，然后直接赋值。`double` 型数据转换为 `float` 型时，通过截取尾数来实现，截断前一般要进行四舍五入操作。例如：

```
float f;
double d=123456789.123456789;
f=d; //f 的值为 123456789.123457
```

 注意：截取的位数和是否四舍五入与具体实现有关。

☐ char 型与 int 型

- `int` 型数值赋给 `char` 型变量时，只保留其最低 8 位，高位部分舍弃。
- `char` 型数值赋给 `int` 型变量时，一些编译程序不管其值大小都作为正数处理，而另一些编译程序在转换时，若 `char` 型数据值大于 127，就作为负数处理。对于使用者来讲，如果原来 `char` 型数据取正值，转换后仍为正值；如果原来 `char` 型值可正可负，则转换后也仍然保持原值，只是数据的内部表示形式有所不同。例如：

```
char c=128; //截取低 8 位
int i=c; //可能是-128
```

两个变量在内存中的表示形式如图 3.6 所示。

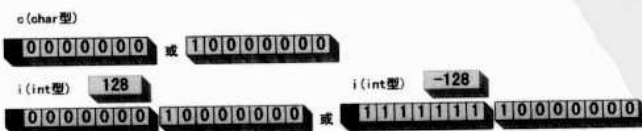


图3.6 变量在内存中的表示



☑ int 型与 long 型

- long 型数据赋给 int 型变量时，将低 16 位值赋给 int 型变量，而将高 16 位截断舍弃（此处假定 int 型占 2 个字节，long 型占 4 个字节）。
- int 型数据赋给 long 型变量时，其外部值保持不变，而内部形式有所改变。即内存中表示位数有所提升。例如：

```
long l=32769;
int i=1;    //i 的值为-1
l=i;        //l 的值为 32769
```

两个变量在内存中的表示形式如图 3.7 所示。



图3.7 变量在内存中的表示

☑ 无符号整数

- 将一个无符号型数据赋给一个占据同样长度存储单元的有符号整型变量时，原值照赋，内部的存储方式不变，但外部值却可能改变。
- 将一个有符号整型数据赋给长度相同的无符号型变量时，内部存储形式不变，但外部表示时总是无符号的。例如：

```
unsigned int u=60000;
int i=u;    //i 的值为-27232
u=i;        //u 的值为 60000
```

两个变量在内存中的表示形式如图 3.8 所示。



图3.8 变量在内存中的表示

C 语言这种赋值时的类型转换形式可能会使人感到不精密和不严格，因为不管表达式的值怎样，系统都自动将其转换为赋值运算符左部变量的类型。而转换后数据可能有所不同，稍不留神就可能带来错误。这确实是个缺点，也遭到许多人批评。产生这样的缺点是因为 C 语言最初是为了替代汇编语言而设计的，所以类型变换比较随意。当然，为了准确地得到自己预期的类型，可以使用下一小节将介绍的强制类型转换。

3.4.3 强制类型转换

在任意一个表达式中使用一个被称为强制类型转换的一元运算符可以指定想要进行的类型转换。



1. 强制类型转换运算符

强制类型转换运算符由括号和类型名组成，其一般形式为：

(类型名)表达式

应该这样理解强制类型转换，在上述表达式中，表达式首先被赋值给以类型名指定类型的某个变量，然后用该变量替换上述的整条语句。应该用想转换成的类型替换类型名，例如，下面的用强制类型转换运算符将 float 型变量 f 转换为 int 型。

```
(int)f //使用强制类型转换将 f 转换成 int 型
```

上面表达式的运算步骤如图 3.9 所示。

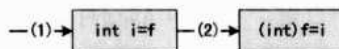


图3.9 强制类型转换过程

2. 强制类型转换符优先级

先来看执行下面程序段后的结果：

```

float result;           //声明存放结果的变量
int dividend,divisor;   //声明除数和被除数
result= dividend/divisor; //进行除法运算
  
```

假设 dividend 的值是 5，divisor 的值是 2，则按照自动类型转换的规则，result 的值为 2.0。此结果显然不是我们预期的结果。要想得到预期的结果 2.5，程序段修改如下：

```
result= (float)dividend/divisor;
```

因为 C 语言将强制类型转换运算符视为一元运算符，其优先级要高于二元运算符，所以编译器会将表达式：

```
(float)dividend/divisor
```

解释为：

```
((float)dividend)/divisor
```

可见，强制类型转换运算符只对其后紧跟的变量 dividend 有效，但是因为 dividend 的类型被转换成了 float 型，按照算术运算的自动转换规则，整个表达式的值也成了 float 型。对此表达式并不造成什么影响。但是有时优先级会引起一些意想不到的结果，例如：

```
(int)f+d
```

其中，f 是 float 型，d 为 double 型。编程者可能预想执行此表达式后得到一个 int 型的结果，但是因为强制类型运算符的优先级高，表达式的结果应该是 double 型。要达到预想结果，应将其改为：

```
(int)(f+d)
```

3. 防止溢出

有时，可以使用强制类型转换来防止溢出。先来看下面的程序段：

```


long l;
int i=10000;
l=i*i;           //可能溢出
  
```




也许你觉得这段程序并没有出错，而其实是被晦涩的自动类型转换给迷惑了！问题就出在表达式 $i*i$ 上，因为两个运算对象都是 `int` 型的， $i*i$ 也应该是 `int` 型的，但是其结果是 100000000，`int` 型明显存储不了这么大的数，所以发生了溢出。幸好可以使用强制类型转换来防止这种溢出，对上面的程序段进行如下修改：

```
l=(long)i*i; //使用强制类型转换防止溢出
```

按照优先级，先执行 `(long)i`，然后执行乘法运算，最后执行赋值，结果是 `long` 型的，不会发生溢出。

 **注意：**如果使用 `l=(long)(i*i)` 将不能防止溢出，因为溢出在强制类型转换前就已经发生了。

在使用强制类型转换时还需注意一点：如 `(float)i+j` 的表达式中，强制类型转换只是改变了此表达式中 `i` 的类型，而不是真正地改变了 `i` 的类型，如果后面还有运算，`i` 的类型依然是其声明时的类型。

3.5 自增和自减运算符

C 语言提供了两个用于自增/自减的运算符，就是本节要介绍的 `(++)` 和 `(--)`。

3.5.1 简化特殊的运算符

因为许多表达式经常需要对变量递增（加 1）或者递减（减 1），例如：

```
i=i+1  
j=j-1
```

自增和自减运算符被提出就为了简化此类操作，简化后可表示为：

```
i++ //或者++i  
j-- //或者--j
```

`++` 和 `--` 运算符虽然简化了操作，但其并非想象中那么简单。其特殊之处在于：它可以用作前缀运算符（用在变量之前，如 `++i`），也可以用作后缀运算符（用在变量之后，如 `i++`）。两种方式都会将变量的值加 1。但其并不相同，区别在于表达式 `++i` 先将 `i` 的值加 1，然后在使用变量 `i` 的值；而表达式 `i++` 是先使用变量 `i` 的值，然后将 `i` 的值加 1。

从运算结果上看，`++i` 的值和运算前 `i` 的值一致，`++i` 的值为运算前 `i` 的值加 1。可以认为，`++i` 是一个“双赢”的运算符，即使 `i` 的值加 1，也使表达式的值加 1。而 `i++` 则是一个“自私”的运算符，只将 `i` 的值加 1，却没有改变表达式的值。

`--i` 和 `i--` 的规则和 `++i` 和 `i++` 相同，只是进行减 1 运算而已。假设 `i` 的初始值为 10，则先执行 `i--` 再执行 `--i` 后变量和表达式的值如图 3.10 所示。

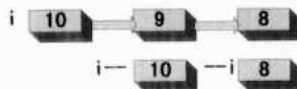


图3.10 `i--` 和 `--i` 的运算过程

3.5.2 使用自增和自减运算符注意事项

使用自增和自减运算符时需要注意以下几点。

- ❑ 自增、自减运算符只能用于变量，而不能用于常量和表达式。例如，下面的表达式都是非法的：

```
++5      //运算对象是常量
(i+j)--  //运算对象是表达式
```

- ❑ 与赋值运算符一样，自增、自减运算符也具有副作用。稍不留神这种副作用就会使程序产生错误。例如：

```
int i=3;
i++;
i*=10000;
```

上面的程序可能会引起溢出，因为执行 `i++` 后，`i` 的值变为 4，而 `i*=10000` 后，`i` 的值为 40000，超出了 `int` 型的最大数 32767。

- ❑ 应避免在一个表达式中对同一个变量多次使用自加、自减运算符。例如，设 `i` 的值为 10，有以下表达式：

```
(i++)+(i++)+(i--)
```

因为 3 个子表达式的求值顺序并不确定，所以此表达式的值可能是 32，也有可能是 30，这完全取决于编译器。

正确的做法是分割这些子表达式为独立的表达式，例如，将上面的表达式分割为：

```
i++
i++
i--
```

- ❑ 应避免将多个包含自增、自减运算符的表达式用于函数的实参。在调用函数时，实参的求值顺序，标准 C 并没有明确规定。例如，`i` 的初值为 10，则下面的格式输出函数输出的值并不是确定的：

```
printf("%d,%d", ++i, i++);
```

有的编译器对实参的求值顺序是从左到右的，所以此输出语句将输出字符串“11,11”，而有些编译器是从右到左的，所以此输出语句将输出字符串“12,10”。要使程序有明确的输出，则需要分割输出语句为：

```
printf("%d", ++i);
printf("%d", i++);
```

- ❑ 前缀自增、自减运算符的优先级和一元的正号、负号优先级相同，且是右结合的。后缀自增、自减运算符的优先级比一元的正号、负号优先级高，且是左结合的。例如 `--i` 相当于 `-(i)`，而 `-j++` 相当于 `-(j++)`。

3.6 逗号运算符与逗号表达式

在 C 语言中，“逗号”也是一种运算符，用逗号将表达式连接起来的式子称为逗号表达式。提供逗号表达式是为了在一个表达式中使用两个或多个表达式。



3.6.1 逗号表达式

逗号表达式的一般形式为：

表达式 1, 表达式 2

其中，表达式 1 和表达式 2 是任意两个表达式，其运算过程可分为以下两步。

第 1 步 计算机表达式 1 的值并不保存此值，而是将其舍弃。

第 2 步 计算机表达式 2 的值，并将其作为整个逗号表达式的值。

此步骤如图 3.11 所示。

正如人们常说的“好戏还在后头”一样，逗号表达式的值也是最后一个表达式的值。需要注意的一点是，表达式 1 有可能具有副作用从而对整个表达式的值产生影响。例如：

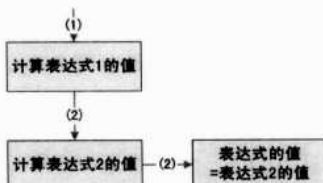


图3.11 逗号运算符计算步骤

`i++, i+j`

显然，首先计算机 `i++` 的值，而后在计算 `i+j` 的值时 `i` 的值已经发生了改变，整个表达式的值受到了表达式 1 的影响。逗号运算符的优先级是所有运算符中最低的，其结合性是自左向右的。例如：

`i+j, j*k`

其等价于：

`(i+j), (j*k)`

逗号表达式也可以嵌套使用，例如：

表达式 1, (表达式 2, 表达式 3)

按照逗号运算符的优先级和结合性，可以将嵌套的逗号表达式扩展为：

表达式 1, 表达式 2, ..., 表达式 *n*

整个表达式的值就是表达式 *n* 的值。

3.6.2 使用逗号表达式注意事项

- ☑ 使用逗号表达式时需要注意以下几点。
- ☑ 程序中使用逗号表达式，通常是要分别求逗号个表达式内各表达式的值，并不一定要求整个逗号表达式的值。例如：

`i=1, j=2`

- ☑ 一般只是完成对 `i` 和 `j` 的赋值，而不希望求表达式的值。
- ☑ 并不是在所有出现逗号的地方都组成逗号表达式。例如下面两种情况虽然出现了逗号，但都不是逗号表达式：

```
int i=1, j=10;           // 变量声明中的逗号
printf("%d%d\n", a, b)    // 函数参数列表中的逗号
```

3.7 关系运算符与关系表达式

正如日常生活中人与人之间会发生各式各样的关系一样，C语言中的各种数据之间也不免要产生关系，表示和处理关系，便要使用本节要介绍的关系运算符和关系表达式。

3.7.1 关系运算符

所谓关系运算符实际上是比较运算，即用来比较两个运算对象，并判断比较的结果是否符合指定的条件。C语言中的关系运算符如表3.3所示。

表3.3 关系运算符

运 算 符	意 义	数 学 符 号
<	小于	<
<=	小于或等于	≤
>	大于	>
>=	大于或等于	≥
==	等于	=
!=	不等于	≠

其中的“==”和“!=”又可以叫做判等运算符。其优先级低于其他4个关系运算符。关系运算符的优先级低于算术运算符，且是左结合的。例如：

```
i+j>=j-k
```

等价于：

```
(i+j)>=(j-k)
```

3.7.2 关系表达式的值

任何类型的表达式都有一个值，关系表达式也不例外。C语言规定，关系表达式的运算结果是一个逻辑值。逻辑值只有两个，分别用“真”和“假”来表示。因为C语言中没有专门的“逻辑值”，所以是用非0表示“真”，用0表示“假”。对于关系运算符来说，如果指定关系成立，其值为1（真），否则为0（假），假设：

```
x=10, y=20
```

则：

```
x<y      //值为1
x>=y     //值为0
x==y     //值为0
x+y<=30  //值为1
```

3.7.3 使用关系运算符注意事项

使用关系运算符需要注意以下几点。

☐ C语言中的关系表示式和数学上的关系表达式不同，有时会产生非预期值。例如：

```
x<y<z
```



并不是验证 y 是否处于 x 和 z 之间，而且等价于：

```
(x<y)<z
```

因为首先执行的子表达式 $x<y$ 的值只有 0 或 1 两种情况，所以只要 $z>1$ ，整个表达式的值就是 1，而和 x 和 y 的具体值没有关系。

- ☐ 如果两个运算对象都是数值型，则按其大小比较。如果两个运算对象都是字符型，则按字符的 ASCII 码值比较大小。如果两个运算对象类型不同，则按常用算术转换的规则进行类型转换。例如：

```
i<l
```

假设变量 i 是 `int` 型的， l 是 `long` 型的。在执行表达式 $i<l$ 时，会将 i 的自动转换为 `long` 型，然后进行比较。

3.8 逻辑运算符与逻辑表达式

在编程中，经常需要将多个表达式连接在一起构成一个表达式完成运算，且表达式的值取决于子表达式的值。显然逗号表达式已经不能“胜任”，从而引出了本节要介绍的逻辑运算符和逻辑表达式。

3.8.1 逻辑运算符

逻辑运算符就是进行逻辑运算的运算符，逻辑运算指运算后只产生真或假两种结果的运算。C 语言中的逻辑运算符如表 3.4 所示。

表 3.4 逻辑运算符

运 算 符	意 义
&&	逻辑与 (AND)
	逻辑或 (OR)
!	逻辑非 (NOT)

因为“!”为一元运算符，所以其优先级要高于“&&”和“||”两个二元运算符，并且“&&”运算符的优先级要高于“||”。即逻辑运算符的优先级顺序为非、与、或。

- 提示：有一个简单的方法记忆此规则，那就是使用其英文的谐音。可用 Not at all (NOT AND OR) 来记忆逻辑运算符的优先级顺序。


3.8.2 逻辑表达式

由逻辑运算符和运算对象组成的合法表达式称为逻辑表达式。逻辑运算的对象可以是 C 语言中任意合法的表达式。逻辑表达式的运算结果和关系表达式的运算结果一样，也是逻辑真 (1) 或逻辑假 (0)。所以，关系表达式可以看成逻辑表达式的一种。两个变量 x 和 y 的逻辑运算规则如表 3.5 所示。

表3.5 逻辑表达式求值

运算表达式	结 果
$x \& \& y$	x 和 y 都不为 0, 结果为 1; 否则, 结果为 0
$x y$	x 和 y 都为 0, 结果为 0; 否则, 结果为 1
$!x$	x 为 0, 结果为 1; x 为 1, 结果为 0

其中, x 和 y 可替换成任意合法的表达式。并且, 只是判断运算对象是否为 0 值 (假), 其他的值都认为是非 0 值 (真)。假设 $x=10$, $y=0$, 则 $x \& \& y$ 为 1, 而 $x || y$ 为 0, $!x$ 为 0。

 **注意:** 逻辑运算符规则可以这样记忆: 非 0 则 1, 全 1 则 1, 全 0 则 0。此规则按照逻辑运算符的优先级顺序表述, 对于非运算, 只有运算对象为 0 (假) 时其值才是 1 (真); 对于与运算, 只有两个运算对象全是 1 (真) 时其值才是 1 (真); 对于或运算, 只有两个运算对象全是 0 (假) 时其值才是 0 (假)。

上节提到 C 语言中的关系表达式和数学中的关系表达式并不相同, 例如 $x < y < z$ 并不代表 y 处在 x 和 z 之间。对于此种情况, 只有采用 C 语言提供的逻辑表达式 $(x < y) \& \& (y < z)$ 才能正确表述此种关系。因为 $\& \&$ 的运算只有当两边表达式值都为真时结果才为真。

3.8.3 “短路”计算

首先说明逻辑运算符的求值顺序是自左向右的, 且运算符 “ $\& \&$ ” 和 “ $||$ ” 的优先级低于关系运算符。所以:

```
a < b & & c == 10 || a != c
```

等价于:

```
((a < b) & & (c == 10)) || (a != c)
```

而且, 逻辑表达式求值时是严格按照从左到右的顺序对子表达式分别求值的, 这样就会引起“短路”计算。即双目逻辑运算符首先计算出左侧运算对象的值, 然后计算右侧运算对象的值。如果逻辑表达式的值可由左侧运算对象单独推导出来, 则不会再计算右侧运算对象的值。例如:

```
7 > 5 || i++
```

因为左侧的表达式值为真, 而对于 “ $||$ ” 运算来说, 全 0 才为假, 所以表达式的值为 1 (真), 右侧的表达式 $i++$ 并没有被执行, 所以 i 的值没有变。可见, 逻辑表达式的“短路”计算有时可以消除运算符的副作用。

3.9 位运算符

C 语言之所以能够像汇编语言一样快速, 并能开发系统软件, 最重要的原因在于它可以直接对位进行运算。本节就来详细的介绍 C 语言中的位运算符。

3.9.1 C 语言的位运算符

在众多软件公司的面试题或者笔试题中, 经常出现这样一道有趣的题目: 用最快的方式计算 2^3 。在数学上, 计算 2^3 只有一种方式: $2 * 2 * 2$, 但是在 C 语言中, 有一种更快的方式,



那就是使用位运算符。此题的答案是：

```
2<<2
```

“<<”叫做左移位运算符，其运算速度要比“*”快得多。C语言共提供了6种位运算符，下面就来认识一下这些“快速”的位运算符，C语言中的位运算符如表3.6所示。

表3.6 位运算符

运 算 符	意 义
&	按位与
	按位或
^	按位异或
~	按位取反
<<	左移
>>	右移

位运算的运算对象只能是整型或字符型数据，不能是其他类型的数据。包含位运算符的表达式的值是一个整型或者字符型的值，而不是逻辑值。“&”、“|”、“^”、“~”又可称为按位运算符，“<<”和“>>”又可称为移位运算符。

“~”是单目运算符，其优先级高于其他5个双目运算符。移位运算符的优先级要高于双目的按位运算符，双目按位运算符“&”、“|”和“^”具有不同的优先级，从高到低的顺序为：“&”、“^”、“|”。

3.9.2 按位与运算符

按位与运算符“&”是将参加运算的两个运算对象，按对应的二进制位分别进行“逻辑与”运算，每个位运算规则如前面所讲，即全1为1，其余为0。例如：

```
z=x&y //x和y按位与并赋值给z
```

假定其中 $x=100$ ， $y=200$ ，则各个变量在内存中表示如图3.12所示。

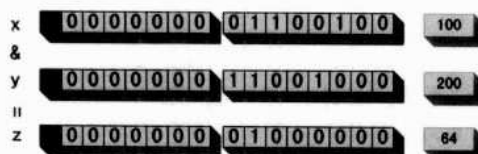


图3.12 变量在内存中的表示

使用按位与运算，可以方便地实现清零或取某个数的某些位，例如：

```
0x00FF&x //取出x的低8位
0xFF00&y //将y的低8位清零
```

第一个表达式将 x 与 $0x00FF$ 进行“&”运算，因为 $0x00FF$ 的二进制数为 0000000011111111 ，其高8位全是0，低8位全是1。所以进行“&”运算后，表达式的值高8位全是0，低8位和 x 的低8位相同。

第二个表达式将 y 与 $0xFF00$ 进行“&”运算，因为 $0xFF00$ 的二进制数为 1111111100000000 ，其高8位全是1，低8位全是0。所以进行“&”运算后，表达式的值高8位和 y 的高8位相同，低8位全是0。



00000000，其高 8 位全是 1，低 8 位全是 0。所以进行 “&” 运算后，表达式的值低 8 位全是 0，高 8 位和 y 的高 8 位相同。

此种方法称为“屏蔽方法”，其中 0x00FF、0xFF00 称为屏蔽字。

3.9.3 按位或运算符

按位或运算符 “|” 是将参加运算的两个运算对象，按对应的二进制位分别进行 “逻辑或” 运算，每个位运算规则如前面所讲，即全 0 为 0，其余为 1，例如：

```
z=x|y //x 和 y 按位或并赋值给 z
```

假定其中 x=100，y=200，则各个变量在内存中表示如图 3.13 所示。

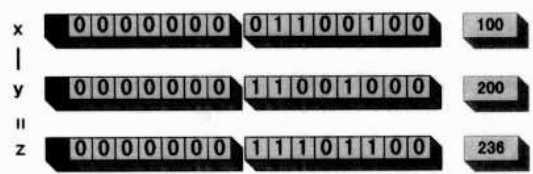


图3.13 变量在内存中的表示

使用按位或运算，可以方便地实现置位，例如：

```
0x00ff|i //将整数 i (2 字节) 的低字节全置为 1
```

其将 i 与 0xff 进行 “|” 运算，因为 0xff 的二进制数为 00000000 11111111，其低 8 位全是 1。所以进行 “|” 运算后，表达式的值低字节全为 1。使用按位或运算符还可以组合新值。例如，要将整数 j 的低字节和整数 k 的高字节组成一个新值，其步骤如下：

- 第 1 步 取出 j 的低字节：0x00ff&j。
- 第 2 步 取出 k 的高字节：0xff00&k。
- 第 3 步 组成新值： 0xff&j|0xff00&k。

注意：不要将按位运算符 “&”、“|” 和逻辑运算符 “&&”、“||” 混淆。前者是对运算对象的每个位进行逻辑与和或运算，其结果是一个整数或字符型数；后者是对整个运算对象进行逻辑与和或运算，其结果是一个逻辑值。

3.9.4 按位异或运算符

按位异或运算符 “^” 也称 XOR 运算符。其将参加运算的两个运算对象，按对应的二进制位分别进行 “按位异或” 运算，每个位运算规则如表 3.7 所示。

表3.7 异或运算规则

表 达 式	结 果
0^0	0
0^1	1
1^0	1
1^1	0



异或运算的规则可以简单地记忆为：相同为 0，不同为 1，例如：

```
z=x^y //x 和 y 按位异或并赋值给 z
```

假定其中 $x=100$, $y=200$, 则各个变量在内存中表示如图 3.14 所示。

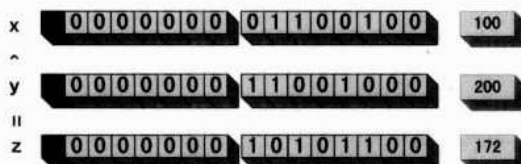


图3.14 变量在内存中的表示

按位异或运算用途广泛，其经常被用于数据加密中，例如：

```
"This is a C program"
```

将此字符串（称为原文）中的每一个字符与字母“a”（密钥）进行按位异或运算，将得到新的字符串（密文）为：

```
5 11A
```

可见，异或后的密文是一堆乱码，这样就可以完成对于原字符串的加密效果。当然，这是最简单的一种加密，非常容易被破解。在实际应用中，会综合运用其他知识和按位异或运算完成对数据的加密。

3.9.5 按位取反运算符

运算符“~”是位运算符中唯一的一个单目运算符，该运算符应该放在运算对象的左边，功能是把运算对象的内容按位取反。即按照二进制位把每位上的 0 变为 1，1 变为 0，例如：

```
y=~x //x 按位取反并赋值给 y
```

假定其中 $x=100$, 则各个变量在内存中表示如图 3.15 所示。




图3.15 变量在内存中的表示

前面所有例子中的数都假定是有符号数，那么，如果 x 是无符号数，结果又是怎样的呢？对于无符号数 x ，其按位取反后的结果根据其数据类型如下：

- ☑ 如果此数是 unsigned char 型，结果为 UCHAR_MAX- x 。
- ☑ 如果此数是 unsigned short 型，结果为 USHRT_MAX- x 。
- ☑ 如果此数是 unsigned int 型，结果为 UINT_MAX- x 。
- ☑ 如果此数是 unsigned long int 型，结果为 ULONG_MAX- x 。

其中的 UCHAR_MAX、USHRT_MAX、UINT_MAX、ULONG_MAX，分别表示 unsigned char、unsigned short、unsigned int、unsigned long int 型数据的最大值。

 **注意：**各种整数类型的最大值和最小值包含在 C 的标准库中，头文件名为 `limits.h`。对于有符号数，包含其最大和最小值，例如，`int` 型的最大值为 `INT_MAX`，最小值为 `INT_MIN`。而对于无符号数，因为其最小值为 0，所以只包含其最大值，例如，`unsigned int` 的最大值为 `UINT_MAX`。

假定上面例子中 `x=100`，并且是 `unsigned int` 型，则各个变量在内存中表示如图 3.16 所示。



图3.16 变量在内存中的表示

可以看出，虽然有符号数和无符号数按位取反时计算的方法有所不同，但是在内存中每个位上的二进制数却是相同的，并且都遵循与原数相反的规则，只不过最高位的含义不同，从而使其外在表现形式不同。

3.9.6 左移运算符

左移运算符“`<<`”是双目运算符，作用是将一个数的各二进制位依次向左移动若干位（由左移位数给出）。运算符左边是移位对象，右边是整型表达式且不能是负数，代表左移的位数，必须少于移位对象的位长。左移时，右端（低位）补 0，左端（高位）移出的部分舍弃，例如：

```
int x=10;
x<<2;
```

执行程序后各个变量和表达式在内存中表示如图 3.17 所示。



图3.17 变量及表达式在内存中的表示

左移时，若左端移出的部分不包括有效二进制数 1，则每左移 1 位，相当于移位对象乘以 2。某些情况下，可以利用左移的这一特性代替乘法运算，以加快运算速度。比如前面所举的计算 2^3 的面试题就是使用此方法，它在内存中的表示如图 3.18 所示。



图3.18 2^3 在内存中的表示

如果左端移出的部分包含有效二进制数 1，这一特性就不适用了，例如：

```
int x=128;
x<<10;
```

执行程序后各个变量和表达式在内存中的表示如图 3.19 所示。



图3.19 变量及表达式在内存中的表示

可以看出, 当 x 左移 10 位后, x 中唯一的 1 位数字 1 被移出了高端, 从而使 $x \ll 10$ 的值变成了 0。

3.9.7 右移运算符

右移运算符“ \gg ”是双目运算符, 作用是将一个数的各二进制位依次向右移动若干位 (由右移位数给出)。右移时, 右端 (低位) 移出的部分舍去, 左端 (高位) 移入的二进制数分两种情况: 对于无符号数和正整数, 高位补 0; 对于负整数, 高位补 0 还是 1 取决于具体实现。有的系统是移入 0, 将其称为“逻辑右移”, 即简单右移; 有的系统是移入 1, 将其称为“算术右移”。

此处用一个程序对右移运算符的规则进行说明, 下面给出程序清单:

```
void main()
{
    int x=100;
    int y=-100;
    unsigned int z=100;
    x >>2; //对正整数 x 右移 2 位
    y >>2; //对负整数 y 右移 2 位
    z >>2; //对无符号数 z 右移 2 位
}
```

执行程序后各个变量和表达式在内存中的表示如图 3.20 所示。

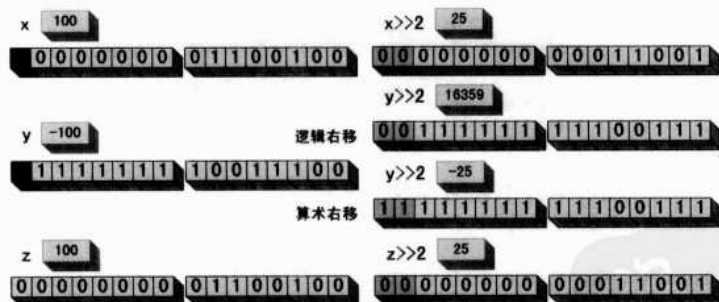


图3.20 变量及表达式在内存中的表示

从图 3.20 中可以看出, 对无符号数或者正整数右移, 以及对负整数算术右移时, 若右端移出的部分不包括有效二进制数 1, 则每右移 1 位, 相当于移位对象除以 2。某些情况下, 可以利用右移的这一特性代替除法运算, 以加快运算速度。

3.9.8 位运算中的整数提升

当位运算中的两个运算对象类型不同时, 一般其位数也会不同。在此情况下, 系统将自动进行如下处理:

- ☑ 先将两个运算对象右端对齐。
- ☑ 再将位数短的一个运算对象向高位扩充。即无符号数和正整数左侧用 0 补全，负整数左端用 1 补全，然后对位数相等的这两个运算数按位进行位运算。例如：

```
int i=100;
long l=200;
i&l;    //对 int 型变量 i 和 long 型变量 l 进行按位与运算
```

执行程序后各个变量和表达式在内存中的表示如图 3.21 所示。

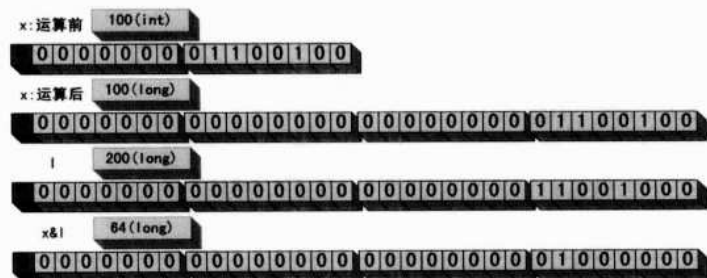


图3.21 变量及表达式在内存中的表示

3.9.9 位运算赋值运算符

位运算符可以和赋值运算符组成复合赋值运算符,C 语言中的位运算赋值运算符如表 3.8 所示。

表3.8 位运算赋值运算符

运 算 符	表 达 式	等 价 于
&=	x&=y	x=x&y
=	x =y	x=x y
^=	x^=y	x=x^y
<<=	x<<=y	x=x<<y
>>=	x>>=y	x=x>>y

提示：赋值运算符的优先级仅高于逗号运算符，使用时要特别注意其求值顺序。例如，表达式 `x<<=y+z` 应该理解为 `x<<=(y+z)`，而不是 `(x<<=y)+z`。

3.10 sizeof 运算符

由于数据占用内存的字节数取决于具体实现，而有时进行的运算涉及运算对象的字节数，所以 C 语言提供了计算数据字节大小的运算符 `sizeof`。本节就对其进行详细介绍。

3.10.1 使用 sizeof

`sizeof` 是 C 语言中求字节数运算符，如同运算符 `++`、`--` 一样是单目运算符。`sizeof` 运算符以字节形式给出其运算对象的存储大小。运算对象可以是一个表达式或括号内的类型名。



运算对象的存储大小由运算对象的类型决定，并且取决于具体实现。

sizeof 可以用于数据类型，也可以用于表达式。

1. 用于数据类型

sizeof 用于数据类型的一般形式为：

```
sizeof(type)
```

其中，数据类型必须用括号括起来。例如：


```
Sizeof(int) //计算 int 型所占的字节数  
Sizeof(double) //计算 double 型所占的字节数
```

2. 用于表达式

sizeof 用于表达式的一般形式为：


```
sizeof(表达式)或 sizeof 表达式
```

表达式可以不用括号括起来，如 sizeof(i)、sizeof i 都是正确形式。带括号的用法清晰且不易出错，所以比较常用。

 **注意：**sizeof 操作符不能用于函数类型、不完全类型或位字段。不完全类型指具有未知存储大小的数据类型，如未知存储大小的数组类型、未知内容的结构或联合类型、void 类型等。这些内容将会在后面的章节中介绍。

3.10.2 sizeof 的结果

sizeof 运算后的结果类型是 size_t 型，该类型能保证容纳实现所建立的最大对象的字节大小。

 **说明：**C 语言中的各种基本数据类型的长度是随平台变化的，为了保证移植性，用一个中间类型 size_t 代表所有长度的类型，sizeof 运算符返回的就是这个类型的数据。在移植时，可以用 typedef 来指定该平台上 size_t 的真实数据类型。比如在 Win32 和 UNIX 平台上，就是 typedef unsigned int size_t，即 size_t 和 unsigned int 一样。

各种类型数据分局进行 sizeof 运算后的结果如下：

- ☑ 若运算对象具有类型 char、unsigned char，其结果等于 1。
- ☑ int、unsigned int、short int、unsigned short、long int、unsigned long int、float、double、long double 类型的大小在 ANSI C 中没有具体规定，则依赖于具体实现，一般可能分别为 2、2、2、2、4、4、4、8、10。
- ☑ 当运算对象是指针时，sizeof 依赖于编译器。一般 UNIX 的指针字节数为 4。
- ☑ 当运算对象具有数组类型时，其结果是数组的总字节数。将在第 6 章“数组与字符串”中详细介绍。
- ☑ 联合类型运算对象的 sizeof 是其最大字节成员的字节数。结构类型运算对象的 sizeof 是这种类型对象的总字节数。将在第 9 章“结构体与共用体”一章中详细介绍。
- ☑ 如果运算对象是函数中的数组形参或函数类型的形参，sizeof 会给出其指针的大小，将在第 5 章“函数”中进行详细介绍。



- ☐ 表达式进行 sizeof 运算后的结果就是其作用于该表达式的值的类型的值。例如（其中 i 为 int 型）：

```
sizeof(i+1)
```

等价于：

```
sizeof(int)
```

3.10.3 sizeof 的优先级

sizeof 的优先级为 2 级，比算术运算符等 3 级运算符优先级高。如果采用不带括号形式的使用方法时要特别注意，例如：

```
sizeof x+1
```

等价于：

```
(sizeof x)+1
```

而不是：

```
sizeof (x+1)
```

sizeof 可以与其他运算符一起组成表达式。例如：

```
i*sizeof (int)
```

其中 i 为 int 类型变量。

3.10.4 各种类型数据长度的计算

由于各种类型数据所占内存的字节数（长度）取决于具体的实现，而在定义数据时需要知道系统中各种类型数据的长度，所以在编程前明确系统中各种类型数据的长度非常必要。下面的程序清单就使用 sizeof 运算符计算系统中各种类型数据的长度：

```
#include "stdio.h"
void main()
{
    printf("系统中 char 型数据占用%d个字节\n",sizeof(char));
    printf("系统中 short 型数据占用%d个字节\n",sizeof(short));
    printf("系统中 int 型数据占用%d个字节\n",sizeof(int));
    printf("系统中 long 型数据占用%d个字节\n",sizeof(long));
    printf("系统中 unsigned short 型数据占用%d个字节\n",sizeof(unsigned short));
    printf("系统中 unsigned int 型数据占用%d个字节\n",sizeof(unsigned int));
    printf("系统中 unsigned long int 型数据占用%d个字节\n",sizeof(unsigned long));
    printf("系统中 float 型数据占用%d个字节\n",sizeof(float));
    printf("系统中 double 型数据占用%d个字节\n",sizeof(double));
    printf("系统中 long double 型数据占用%d个字节\n",sizeof(long double));
}
```

3.11 语 句

程序其实是由若干条数据定义和语句组成的。第 2 章已经介绍了数据定义，本节就来介绍 C 语言中的语句。



3.11.1 什么是语句

语句是构成程序的基本成分。程序是一系列带有某种必需的标点的语句集合。在 C 语言中，这个标点就是分号“;”，例如：

```
number=100
```

这是一个表达式，可能是一个较大的语句的一个组成部分，但不是一条语句。而：

```
number=100;
```

是一条语句。一条语句必须是一条完整的计算机指令。如上面的例子，C 语言把任何后面带有一个分号的表达式看作一条语句，准确地说，是一条表达式语句。所以，下面的两行也是语句：

```
100;  
123+456;
```

但是，这些语句对于程序来说没有什么实质作用。一般情况下，有用的语句可以改变值或者调用函数，例如下面的两条语句：

```
number++;  
y=sqrt(x);
```

第一条语句通过“++”运算符将变量 `number` 的值改变为 `number+1`，而第二条语句通过调用 `sqrt()` 函数求得变量 `x` 的平方根并赋值给 `y`。一条语句是一条完整的指令，但是一条完整的指令并不一定就是一条语句。例如下面的语句：

```
y=100-(x++);
```

在这条语句中，子表达式 `x++` 是一条完整的指令，但其只是一条语句的一部分。可以看出，是否是一条完整的指令不是判别是否是一个语句的唯一标尺，分号也是一个标尺。那么，只要以分号结束的一段程序是否就一定是一条语句呢？下面的这行是否是一条语句呢？

```
int i;
```

此处要说明的是，程序包括数据描述和数据操作。数据描述主要定义数据结构（用数据类型表示）和数据初值，由数据定义来实现。数据操作对已提供的数据进行加工，这部分才是由语句来实现的。

所以，只有出现在数据操作部分的带有分号的完整的指令才能叫做语句，而上面的程序明显是处于数据描述部分，不能叫做语句，而将其叫做数据定义。

3.11.2 语句类型

C 语句（C 语言中的语句）有五种类型：表达式语句、函数调用语句、控制语句、空语句和复合语句。

1. 表达式语句

表达式语句由表达式加上分号“;”组成。其一般形式为：

表达式；

执行表达式语句就是计算表达式的值，例如：

```
y=1+x;           //赋值语句  
i++;             //自增1语句，i 值增1  
x+y;            //加法运算语句，但计算结果不能保留，无实际意义
```




2. 函数调用语句

函数调用语句由函数名、实际参数加上分号“;”组成。其一般形式为：

函数名(实际参数表)

执行函数调用语句就是调用函数体并把实际参数赋予函数定义中的形式参数，然后执行被调函数体中的语句，求取函数值，例如：

```
printf("Welcome to C world"); //调用标准格式输出库函数，输出字符串
```

有关函数调用语句的细节将在第5章“函数”中介绍。

3. 控制语句

控制语句用于控制程序的流程，以实现程序的各种结构方式。控制语句由特定的语句定义符组成。C语言有9种控制语句，可分成以下3类：

- ☑ 条件判断语句：if 语句、switch 语句。
- ☑ 循环执行语句：while 语句、do while 语句、for 语句。
- ☑ 跳转语句：break 语句、continue 语句、goto 语句、return 语句。

有关控制语句的细节将在下一章中介绍。

4. 空语句


只有分号“;”组成的语句称为空语句。空语句是什么也不做的语句。在程序中空语句可用来做空循环体。

5. 复合语句

把多个语句用花括号“{}”括起来组成的一个语句称为复合语句，也称为语句块或者块语句。其格式为：

```
{  
    语句 1;  
    语句 2;  
    ...  
    语句 n;  
}
```

在C程序中应把复合语句看成是一条语句，而不是多条语句。复合语句用于任何只能出现一条简单语句的地方。

 **注意：**复合语句是一个特例，它是唯一不以分号“;”结尾的语句，但复合语句内的各条语句都必须以分号“;”结尾。

3.11.3 赋值语句

1. 赋值语句的解释

由赋值表达式加上分号构成的表达式语句叫做赋值语句。其功能和特点都与赋值表达式相同。赋值语句是程序中使用最多的语句。例如，下面是一条赋值语句：

```
y=10+x;
```



2. 使用赋值语句注意事项

使用赋值语句时应注意以下几点。

☑ 由于在赋值符“=”右边的表达式也可以是一个赋值表达式，因此，下述形式：

```
变量=(变量=表达式);
```

是成立的，从而形成嵌套的赋值语句，将其展开之后的一般形式为：

```
变量=变量=...=表达式;
```

例如：

```
a=b=c=d=e=100;
```

按照赋值运算符的右接合性，上面的赋值语句等效于下面的 5 条语句：

```
e=100;
d=e;
c=d;
b=c;
a=b;
```

☑ 注意在变量说明中给变量赋初值和赋值语句的区别。

给变量赋初值是变量说明的一部分，不属于语句，赋初值后的变量与其后的其他同类变量之间仍必须用逗号间隔，而赋值语句则必须用分号结尾。例如，下面的语句是在给变量赋初值：

```
int a=100,b,c;
```

下面的语句是两个单独的赋值语句。

```
a=200;
b=300;
```

☑ 在变量说明中，不允许连续给多个变量赋初值。例如下述变量说明是错误的：

```
int a=b=c=100;
```

必须写为：

```
int a=100,b=100,c=100;
```

而赋值语句允许连续赋值。例如下面的赋值语句是正确的：

```
a=b=c=200;
```

☑ 注意赋值表达式和赋值语句的区别。赋值表达式是一种表达式，可以出现在任何允许表达式出现的地方，而赋值语句则不能。

下述语句是合法的：

```
if((x=y+100)>0) z=x;
```

表示若表达式 $x=y+100$ 大于 0，则 $z=x$ 。下述语句是非法的：

```
if((x=y+100;)>0) z=x;
```

因为 $x=y+100$ 是语句，语句是不能出现在表达式中的。



第4章

流程控制



所谓计算机语言的流程就是程序中每个操作执行的先后顺序。通常，一门计算机语言会提供 3 种形式的程序流程：顺序、分支和循环。这 3 种流程叫做基本控制流程。基本控制流程有一个特点：具有一个入口和一个出口。此特点提高了程序的清晰度和易读性。除此之外，C 语言中还提供了跳转结构。但因其通常是配合 3 种基本控制流程使用，故从严格意义上来说，不能算是一种控制流程。本章为了让读者对流程控制有一个整体的了解，将其视为一种单独的控制流程来介绍。本章将主要围绕顺序、分支、循环和跳转这几种程序流程来展开讲述。





4.1 流程的表示方法

在实际应用中，可以用许多不同的方法来表示流程。本节介绍其中最常用的两种：自然语言表示法和流程图表示法。

4.1.1 自然语言表示法

对于控制流程的表示，人们首先想到的是自然语言。自然语言就是人们日常生活中使用的语言，例如汉语、英语或者其他语言。用自然语言表示程序流程非常易懂，例如下面是用自然语言来表示求两个数中最大数的程序流程。

第 1 步 定义两个变量 x 、 y 存放两个要比较的数。

第 2 步 定义一个变量 z 存放两个数中的最大数。

第 3 步 用户输入 x 、 y 的值。

第 4 步 判断 x 的值是否大于 y 。

a. 如果满足条件则将 x 的值赋给 z 。

b. 如果不满足条件则将 y 的值赋给 z 。


第 5 步 输出 z 的值。

可以看出，用自然语言来表示程序的流程比较烦琐，一个很简单的程序需要用很长一段的文字。此外，自然语言还容易引起歧义。例如，有这样一句话：“小李看见小张在开他的车”。仅凭这句话无法分辨其中的“他”是小李还是小张。所以，一般情况下，很少使用自然语言来表示程序的流程。

4.1.2 流程图表示法

流程图是用一些图形框来代表各种操作，从而表示过程控制流程的一种图形。与自然语言表示法相比，流程图表示法更加容易理解。

流程图的作用是描述人们解决问题的方法、思路或算法。

 **注意：**本章中所提到的流程图都是指基本流程图，如果了解其他形式的流程图，请参考相关资料。

美国国家标准化协会 ANSI (American National Standard Institute) 规定了一些常用的流程图符号，如图 4.1 所示。

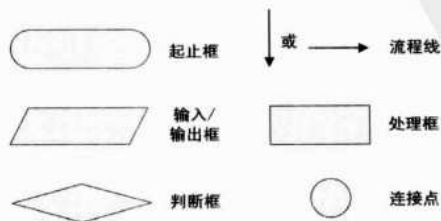





图4.1 常用流程图符号

各种符号的作用和流程线的数目如表 4.1 所示。

表4.1 流程图符号作用表

符号图形	符号名称	作 用	流 程 线
	起始、终止框	表示开始和结束	起始框：一条流出线 终止框：一条流入线
	输入/输出框	框中标明输入/输出的内容	一条流入线和一条流出线
	处理框	框中标明进行什么处理	一条流入线和一条流出线
	判断框	框中标明判定条件并在框外标明判定后两种结果的流向	一条流入线和两条流出线，但只有一条流出线起作用
	连接点	连接两段流程线	两条流程线，无流入和流出之分

同样是求两个数 x 和 y 中的最大数，其流程图如图 4.2 所示。

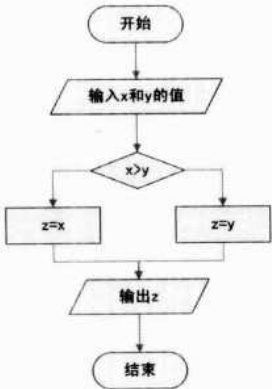


图4.2 求最大数的流程图

可见，用流程图表示程序的控制流程有以下几个优点：

- ☑ 采用简单规范的符号，画法简单。
- ☑ 结构清晰，逻辑性强。
- ☑ 便于描述，容易理解，且不容易引起歧义。

所以，大多数情况下，都推荐使用流程图来表示程序的控制流程。

4.2 顺序结构

顺序结构是一种最简单且使用最频繁的控制流程。读者对其应该并不陌生，前面出现的大多数程序都是这种结构的。顺序就是程序的逻辑次序，按照语句的顺序将其组合起来就构成了顺序结构的程序。

4.2.1 什么是顺序结构

顺序结构就是顺序执行程序的各种操作，即按照操作出现的先后次序依次执行，即前面所说的直线型程序。如果程序中存在两个操作 A 和 B ，执行完 A 操作后，执行 B 操作。如图 4.3 所示。



要理解顺序结构，不妨回忆一下在语文课上做过的句子组合练习。所谓句子组合练习，就是给出若干个分句，然后按照分句的逻辑关系将其组合成一个完整的句子，例如：

- 第 1 步** 冰箱出了问题。
- 第 2 步** 冰箱又能正常工作了。
- 第 3 步** 厂家派维修人员进行修理。
- 第 4 步** 小李给厂家打电话报修。
- 第 5 步** 小李买了台冰箱。

分析上面的 5 个分句可以得出一个规律，即存在先后顺序，按照先后顺序将其组合成一个句子是：小李买了台冰箱，冰箱出了问题，小李给厂家打电话报修，厂家派维修人员进行修理，冰箱又能正常工作了。此事件的过程如图 4.4 所示。



图4.3 顺序结构

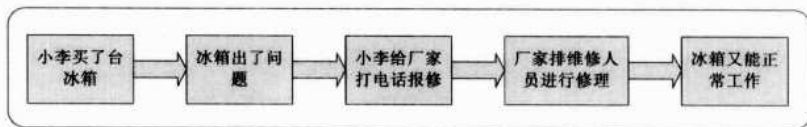


图4.4 修理冰箱过程图

按照这种思路，C 语言的顺序结构就是把类似于分句的结构（语句）按照某种顺序组合，从而形成一个完整的程序或程序段。

4.2.2 顺序结构程序设计方法

将多个具有顺序的分句按照其顺序组合就构成了句子。同样，将各条语句按照某种逻辑上的次序组装起来，产生一个完整的程序或者程序段就成为顺序结构的程序或者程序段。如果要和计算机进行交互，还需要加上输入/输出。组装具有顺序的语句构成顺序结构程序的方法有很多种，此处介绍一种简单有效的方法：

- ☑ 按照题目要求定义若干变量和常量，并写出若干条语句。相当于组合句子中的分句。
- ☑ 按照题目要求画出程序的流程图，流程图就是顺序结构的顺序。需要特别提醒的是：有些程序可能是非常简单的，画流程图似乎显得比较麻烦，但是画流程图非常有利于整理自己的编程思路，使自己在编程时逻辑清晰，所以还是要养成画流程图的好习惯。
- ☑ 最后根据流程图将数据定义和各条语句进行组合，并加上需要包含的头文件等其他内容，形成完整的程序。

当然，上述前两条的次序并非固定，如果你觉得先画流程图更容易理清题目的逻辑关系，可以先画流程图，再编写语句。

下面就通过一个实例来演示顺序结构程序设计的基本方法。

【题目】设计并实现一个计算三角形面积的程序。三角形的3条边的长度由用户来指定，程序需要输出三角形的面积。

【分析】假设三角形的3条边长为 a 、 b 、 c 。由数学知识知道求三角形面积的公式为：

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$

其中， $s = \frac{1}{2}(a+b+c)$ 。

第1步 定义变量和写出语句。

首先，要定义需要用到的变量。由于边长数据有可能为小数，所以定义3个 float 型变量 a 、 b 、 c 来存放边长。同时定义一个 float 型变量 s 来存放中间结果，也就是 $1/2(a+b+c)$ 。还要定义一个变量 area 来存放面积，因为面积的数值有可能会很大，所以将其定义为 double 型。

通过计算公式，可以写出求面积的语句为： $\text{area}=\text{sqrt}(s*(s-a)*(s-b)*(s-c))$ ；其中， $\text{sqrt}()$ 为求平方根函数，是一个库函数。在 C 语言库中已经为编程人员写好了许多常用的数学函数，在编程时，最好调用这些库函数。这样做的好处是既可以节省时间，又可以防止自己编程时出错。使用这些数学函数时必须包含头文件 math.h 。

同时，在求面积的语句中出现了 s ，同样也要写出语句 $s=1.0/2*(a+b+c)$ ；。还有，按照题目的要求，3条边的长度是由用户输入的，由于这些边长都是数字，所以选择格式输入函数 $\text{scanf}()$ 来进行输入。输入语句为： $\text{scanf}("%f%f%f"&a,&b,&c)$ ；。最后，按照题目的要求，要将计算出的面积进行输出。选择格式输出函数 $\text{printf}()$ 来进行输出。在输出时，假设要保留小数点后两位小数，并且宽度为10，则输出的语句为： $\text{printf}("area=\%10.2f\n",\text{area})$ ；。

第2步 画程序流程图。

根据流程图的知识 and 题目的要求，画出这个题目的流程图如图4.5所示。



图4.5 程序流程图

**第3步** 组合语句，编写程序。

按照流程图中的顺序，组合在第一步中定义好的变量和写出的语句，得到根据用户输入的三角形的3条边的长度，计算并输出三角形面积的程序。程序清单如下所示：

```
#include <stdio.h>
#include <math.h>

void main()
{
    float a,b,c,s;           //定义存放三角形边长以及存放中间结果的变量
    double area;             //定义存放面积的变量
    scanf("%f%f%f",&a,&b,&c); //输入边长
    s=1.0/2*(a+b+c);         //计算中间结果
    area=sqrt(s*(s-a)*(s-b)*(s-c)); //求三角形面积
    printf("area=%10.2f\n",area); //输出三角形面积
}
```

顺序结构的程序一般都比较简单，一般的程序都不会单单是顺序结构的，顺序结构出现更多的形式是一个程序中的某个或某几个程序段。当程序中出现两条以上在顺序上没有先后关系的语句时，在程序执行时往往是选择性地执行某一条语句，这就是下一节要介绍到的分支结构。

4.3 分支结构

分支是3种基本流程之一，在大多数程序中都会包含这种结构。如果程序中出现了不先在先后顺序的语句并伴随着一些选择条件，要使用这些语句构成符合某种要求的程序，顺序结构就无法对其进行处理，从而引入了分支结构。

C语言中能够实现分支结构的形式有：if语句、条件运算符和switch语句等。本节将从这3个方面来详细介绍C语言中的分支结构。

4.3.1 什么是分支结构

分支结构就是通过进行一个判断，在两个可选的语句序列之间进行选择执行。例如，根据判断条件是否成立选择执行A操作或者B操作，如图4.6所示。

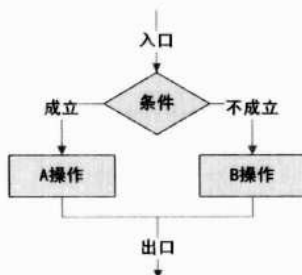


图4.6 分支结构

要理解分支结构，还是看一下句子组合练习。不过，此处的组合练习中没有先后顺序的分句和一些判断条件，例如：

- ☑ 人的正常体温是 $36^{\circ}\text{C}\sim 37^{\circ}\text{C}$ 。
- ☑ 体温在 $37.3^{\circ}\text{C}\sim 38^{\circ}\text{C}$ 之间叫低烧。
- ☑ 体温在 $38.1^{\circ}\text{C}\sim 40^{\circ}\text{C}$ 之间叫高烧。

第一个分句说明了一种正常的情况，第二个和第三个分句中的条件就不同于第一个分句，并且这两个分句的条件也不同。不难看出，组合后的句子为：人的正常体温是 $36^{\circ}\text{C}\sim 37^{\circ}\text{C}$ ，当高于这个温度时，一般就发烧了。发烧分为两类，体温在 $37.3^{\circ}\text{C}\sim 38^{\circ}\text{C}$ 之间为低烧，体温在 $38.1^{\circ}\text{C}\sim 40^{\circ}\text{C}$ 之间为高烧，如图 4.7 所示。

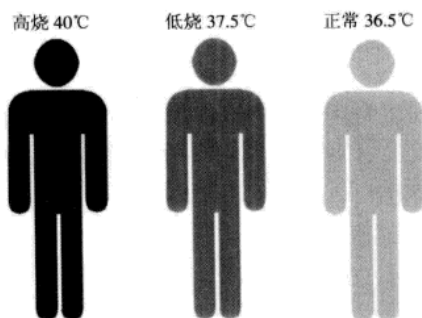


图4.7 不同的体温

按照这种思路，C 语言的分支结构就是先对某种条件进行判断，然后按照是否满足条件去执行顺序相同的两个语句中的一个。当然按照分支结构组合语句后，形成的可以是一个完整的程序，也可以是程序段。

在具体实现方式上，C 语言分支结构中选择过程是通过某种选择语句来实现的，选择条件一般就是前面章节学过的逻辑表达式或条件表达式，选择后执行的语句可以是单条语句，也可以是复合语句。

4.3.2 if 语句的解释

if 语句被称为选择语句或者分支语句，是用来判定所给定的条件是否得到满足，根据判定结果的真或假来决定执行给出的两种操作中的某一种。判断结果真或假是看条件表达式的值是非 0 值还是 0 值，如果条件表达式的值是非 0 值，则判断结果就为真；如果条件表达式的值是 0 值，则判断结果为假。

4.3.3 if 语句的 3 种形式

1. 基本形式

基本形式 if 语句的一般形式为：

```
if(表达式) 语句;
```

其语义是：如果表达式的值为真，则执行其后的语句；如果表达式的值为假，则不执行该语句。其流程图如图 4.8 所示。

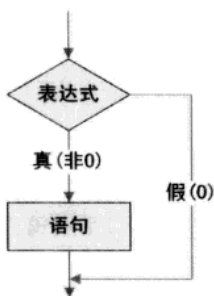


图4.8 基本形式流程图

通常，表达式是一个关系表达式，作用就是比较两个量的大小，像表达式 $x > y$ 或 $z == 6$ 。如果表达式的值为真，像上面表达式，就是 x 大于 y 或者 z 等于 6，则执行语句；否则，忽略语句。

在一般情况下，表达式可以是任何形式的表达式，表达式的值为 0 就被视为假。语句部分可以是单条语句，也可以是复合语句，或称代码块，要用花括号括起来。简单语句和复合语句使用方法程序清单如下所示：

```
#include <stdio.h>

void main()
{
    int i;
    scanf("%d",&i);
    if(i>100)
        printf("i 的值大于 100\n");    //简单语句
    if(i<=100)
    {
        i=100;
        printf("i 的值现在等于 100");
    }
}
```

在上面的程序中，第一个 if 语句后面出现的是一条简单语句，完成输出提示信息“i 的值大于 100”。第二个 if 后面出现的是一条复合语句，首先将 i 重新赋值为 100，然后输出提示信息“i 的值现在等于 100”。

2. if...else 形式

if...else 形式的一般形式为：

```
if(表达式)
    语句 1;
else
    语句 2;
```

其语义是：如果表达式的值为真，则执行语句 1；如果值为假或零，则执行跟在 else 后面的语句 2。语句可以是简单语句或复合语句。C 语言本身是没有要求缩排的，但是缩排是一种标准风格，可以使语句依赖于判断而执行这一事实显得清晰。

其流程图如图 4.9 所示。

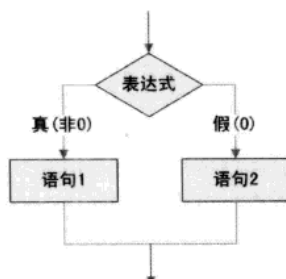


图4.9 if-else形式流程图

基本形式的 if 语句使你能够选择是否执行某个动作，而 if-else 形式的 if 语句使你可以在两个动作间直接进行选择。如果需要选择的动作有两个，并且其判断条件正好相悖，if...else 语句将是较好的选择。例如，简单语句和复合语句使用方法程序正好是符合这种条件的一个程序，下面的程序清单是用 if...else 形式的实现：

```
#include <stdio.h>

void main()
{
    int i;
    scanf("%d",&i);
    if(i>100)                //判断条件
        printf("i 的值大于 100\n"); //条件为真时执行的语句
    else
    {                          //条件为假时执行的语句
        i=100;
        printf("i 的值现在等于 100");
    }
}
```

比较两个程序，基本形式的程序中使用了两个 if 语句。而这两个语句的判断条件正好是相悖的。在 if...else 形式的程序中，判读条件为 $i>100$ ，在这个条件满足时，执行 if 后面的语句，条件不满足时，也就是 $i\leq 100$ 时，执行 else 后面的语句块。可以看出，在实现这种需求的程序时，if...else 的结构更加清晰。

3. else if 形式

else if 形式的一般形式为：

```
if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
...
else if(表达式 m)
    语句 m;
else
    语句 n;
```



其语义是：依次判断表达式的值，当出现某个值为真时，则执行其对应的语句，然后跳到整个 if 语句之外继续执行程序。如果所有的表达式均为假，则执行语句 n，然后继续执行后续程序。

else if 形式的流程图如图 4.10 所示。

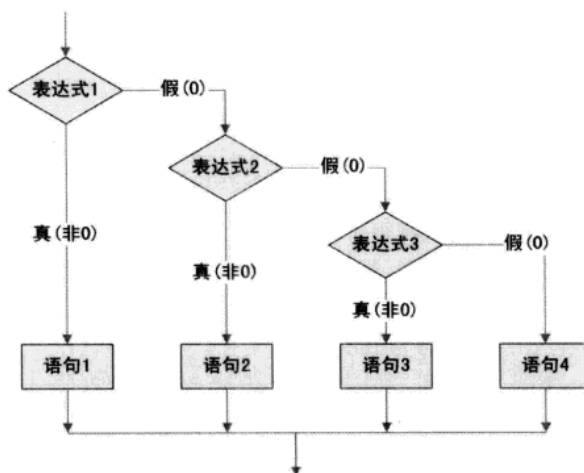


图4.10 else if形式流程图

“else if”形式的 if 语句用于有两个以上选择（多重选择）的情况。在现实生活中，这种情况有很多。例如，许多网吧的计费系统就是一个多重选择的例子。假设某个网吧是以时间段上网人数的多少为标准进行计费的，在早上 8 点到中午 12 点这段时间的计费方式是每小时 1 元，在中午 12 点到下午 6 点的计费方式是每小时 1.5 元，而到了下午 6 点到晚上 12 点的计费方式是每小时 2 元。要用程序实现这个系统的话，else if 形式是 3 种 if 语句形式中最好的选择。

假设每个时段的上网人数和平均时间如表 4.2 所示。

表4.2 上网分布表

时 间 段	人 数	平 均 时 间
8: 00 - 12: 00	50	3
12: 00-18: 00	100	5
18: 00-24: 00	200	5.5

计算每个时段上网费用的程序清单如下：

```
#include <stdio.h>

void main()
{
    int time;           //上网的时间
    float rage;         //上网的费用
    printf("请输入上网的时间\n");
```



```
scanf("%d",&time); //输入上网时间
if(time<12)
    rage=50*3*1;
else if(time<18)
    rage=100*5*1.5;
else
    rage=200*5.5*2;
printf("此阶段上网产生的费用是: %d",rage);
}
```

在上面的程序中,第8行语句首先接收输入的上网时间并存放在变量 `time` 中,接下来执行第9行的 `if` 语句,如果满足条件 `time<12`,则执行第10行语句,然后跳出 `if` 语句,再执行第15行的语句,程序结束。如果不满足这个条件,则执行第11行的 `else if` 语句,如果满足条件 `time<18`,则执行 `else if` 后面的第12行语句,然后跳出 `if` 语句,再执行第15行的语句,程序结束。如果仍然不满足这个条件,则执行 `else` 后面的第14行语句,然后跳出 `if` 语句,直至程序结束。

4.3.4 嵌套的 if 语句

当 `if` 语句中的执行语句又是 `if` 语句时,则构成了嵌套的 `if` 语句。

其一般形式为:

```
if(表达式)
    if 语句;
```

或者为:

```
if(表达式)
    if 语句;
else
    if 语句;
```

内嵌的 `if` 语句可以是上述 `if` 语句3种形式的任意一种,当内嵌的 `if` 语句是 `if...else` 型时,将会出现多个 `if` 和多个 `else` 重叠的情况,这时要特别注意 `if` 和 `else` 的配对问题,例如:

```
if(表达式1)
if(表达式2)
    语句1;
else
    语句2;
```

此处出现了两个 `if` 和一个 `else`,其中的 `else` 究竟是与哪一个 `if` 配对呢?是应该理解为:

```
if(表达式1)
    if(表达式2)
        语句1;
    else
        语句2;
```

还是应该理解为:

```
if(表达式1)
    if(表达式2)
        语句1;
```



```
else  
    语句 2;
```

可以看出, 这样的程序存在二义性。为了避免这种二义性, C 语言规定, `else` 总是与其前面最近的 `if` 配对, 因此上面的例子应按第一种假设理解。

上一个程序用 `else if` 形式的 `if` 语句实现了一个网吧的计费系统。用内嵌的 `if` 语句也能实现这个程序。两种方法的复杂程度相当, 具体在编程中选择那种方法完全取决于你的习惯。用内嵌的 `if` 语句实现网吧计费系统程序清单如下所示:

```
#include <stdio.h>  
  
void main()  
{  
    int time;           //上网的时间  
    float rage;         //上网的费用  
    printf("请输入上网的时间\n");  
    scanf("%d",&time); //输入上网时间  
    if(time<12)  
        rage=50*3*1;  
    else  
    {  
        if(time<18)  
            rage=100*5*1.5;  
        else  
            rage=200*5.5*2;  
    }  
    printf("此阶段上网产生的费用是: %d",rage);  
}
```

4.3.5 应用 if 语句注意事项

1. 应注意的问题

因为 `if` 语句的形式多样, 而且可以嵌套使用, 使 `if` 语句变得有些复杂。在使用时, 应该注意以下问题。

- 在 3 种形式的 `if` 语句中, 在 `if` 关键字之后均为表达式。该表达式通常是逻辑表达式或关系表达式, 但也可以是其他表达式, 如赋值表达式等, 甚至也可以是一个变量。例如:

```
if(x=100) 语句;  
if(y) 语句;
```

这样的语句都是允许的。只要表达式的值为非 0, 即为真。例如上面的第一行语句中表达式的值永远为非 0, 所以其后的语句总是执行的, 当然这种情况在程序中不一定会出现, 但在语法上是合法的。

又例如下面的程序段:

```
if(x=y)  
    printf("%d",x);  
else  
    printf("x=0");
```


这条语句的语义是：把 y 的值赋予 x，若为非 0 则输出该值，否则输出字符串"x=0"。事实上，只有 y 的值为 0 时，表达式的值为 0，程序执行 else 部分，其他的情况下，程序都执行 if 部分。这种用法在程序中是经常出现的。

- ☑ 在 if 语句中，条件判断表达式必须用括号括起来，后面不能加分号，因为它是一个表达式，而在 if 或者 else 后面的语句之后必须加分号，因为它是语句。例如：

```
if(x==y);  
    printf("x 和 y 的值相等");
```

是不对的，正确的形式应该是：

```
if(x==y)  
    printf("x 和 y 的值相等");
```

- ☑ 在 if 语句的 3 种形式中，所有的语句应为单个语句，如果要想在满足条件时执行一组（多个）语句，则必须把这一组语句用“{}”括起来组成一个复合语句。但要注意的在“}”之后不能再加分号。例如：

```
if(a>b)  
    t=a;  
    a=b;  
    b=t;  
else  
    ...
```

在上面程序段中，编程者的想法可能是：在 a 大于 b 的情况下，将 a 和 b 的值交换，从而让 a 的值小于 b 的值。但是 if 语句在进行判断后，如果 a 大于 b，则程序执行完 if 后面的语句 t=a; 后就应该结束了，但是这条语句后面还有两条语句，所以这个程序在编译时会出错。

要实现预想的功能，可以将程序段改动如下：

```
if(a>b)  
{  
    t=a;  
    a=b;  
    b=t;  
}  
else  
    ...
```

- ☑ 不要设想用缩排的方式来按照自己的预期对内嵌的 if 和 else 配对，因为缩排只是为了实现程序的标准化，使程序看起来清晰，并不会影响程序的执行顺序。正确的方式是使用“{}”。例如：

```
if(a<10)  
    if(a>1)  
        t=1;  
else  
    t=2;
```

在上面的程序段中，编程者预想的功能是想通过缩排的方式让第一个 if 和 else 进行配对，但结果并非所想。假设 a 的值为 100，第一个 if 语句的条件 a<10 不满足，程序将执行第二个 if 语句。其条件 a>1 被满足，所以 t=1。但是按照编程者的预想，因为不满足第一个 if 的



条件，所以应该执行 else 部分，设想的结果为 t=2。要想得到这个结果，可以将程序改动如下：

```
if(a<10)
{
    if(a>1)
        t=1;
}
else
    t=2;
```

2. 建议

为了在使用 if 语句编写程序时不出错或者少出错，以及使编写的程序有较好的易读性，提出以下建议：

- ☑ 如果使用的是 if...else 形式、else if 形式、嵌套形式的 if 语句，并且选择的两个操作有难易之分。例如，某个操作用简单语句就可以完成，某个操作要用复合语句才能完成。建议将简单的操作放在 if 部分、将复杂的操作放在 else 或者 else if 部分。

下面的程序段就遵循了这种原则：

```
if(a>b)
    t=a;
else
{
    if(a<b)
        t=b;
    else
        t=a;
}
```

这样做有以下两点好处：

- 降低编程难度。首先，因为简单操作的难度低，在 if 部分进行了处理；然后在 else 部分，可以将复杂操作再进行细分，使其变成若干个简单操作，如果这些简单操作是顺序结构，那么依次排列即可；如果这些操作仍然是分支结构，再按照这种原则循环地编程即可。
- 提高程序的运行速度。采用这种方式编程，如果当 if 中的条件得到满足，程序只需要执行 if 后面的简单操作即可。即使 if 条件不满足，如果 else 部分的操作仍然是分支结构的，那么，也能够保证程序执行的语句最少，从而提高程序的运行速度。
- ☑ 如果 if 语句的某个部分后面的语句为简单语句，在语法上没有要求将这个语句用“{}”括起来，但是出于程序结构的清晰性，以及减少编程错误，建议将 if 语句的每个部分后面的语句都用“{}”括起来。例如，下面的程序段就遵循了这种原则：

```
if(x>=0)
{
    printf("x 是一个正数");
}
else
{
    printf("x 是一个负数");
}
```

这样做出于以下两点考虑:

- 这些多出来的程序行并不会影响程序的运行速度。因为编译器在编译程序时,会将其去掉。
- 这样的程序会给编写和阅读程序提供很大的帮助。当编写程序时,可以采用这样的形式先将程序的框架搭起来,不需要考虑 if 或者 else 后面到底是简单语句还是复合语句。然后再往“{}”里面填写程序,这样既不容易出错,又可以一边编写程序一边整理自己的思路,从而提高编程速度和降低程序的错误率。当阅读程序时,想知道 if 部分做什么,只需要查看和 if 对齐的“{}”即可,想知道 else 部分做什么,只需要查看和 else 对齐的“{}”即可。特别是在阅读复杂程序时,帮助非常大。

☑ 建议使用类似于 if(0==x)的形式。

这个形式看上去很奇怪,采用这么奇怪的形式,其实是用来防止一个常见错误的小技巧。

例如:

```
if(x=0)
```

这样的程序并没有出错,但是你可能想做的事情是判断 x 是否等于 0,而不是把 0 赋值给 x。如果你养成了把常量写在前面的习惯,那么你就会把上面的程序写成:

```
if(0=x)
```

这程序是无法编译通过的。这样就可以有效地防止将“==”写成“=”的错误。当然,这样的程序确实不怎么好看,所以有人建议应该让编译器对 if(x=0)的形式进行警告。事实上,现在有很多的编译器会对条件中的赋值进行警告。

4.3.6 条件运算符的解释

在开始介绍条件运算符之前,先来看一个求两个数中较大数的程序,程序清单如下所示:

```
#include <stdio.h>

void main()
{
    int x,y,max;
    printf("\n input two numbers: ");
    scanf("%d%d",&x,&y);
    if(x>y)
    {
        max=x;
    }
    else
    {
        max=y;
    }
}
```

此程序结构清晰,并能完成预期功能。不过,仅仅是实现求两个数中的最大数这样简单的要求,程序看起来却有些复杂。其实,有一种更加简洁的方式来完成这种要求,那就是条件运算符。



通过前面章节对运算符的介绍可以知道，有一个操作数的运算符叫做一元运算符，有两个操作数的运算符叫做二元运算符。按照该惯例，有三个操作数的运算符就叫做三元运算符，条件运算符就是 C 语言中唯一的一个三元运算符。

条件运算符由“?”和“:”组成，由条件运算符组成的条件表达式的一般形式为：

表达式 1? 表达式 2: 表达式 3

其语义是：如果表达式 1 的值为真（非 0），则以表达式 2 的值作为条件表达式的值；如果表达式 1 的值为假（0），以表达式 3 的值作为整个条件表达式的值。其流程图如图 4.11 所示。

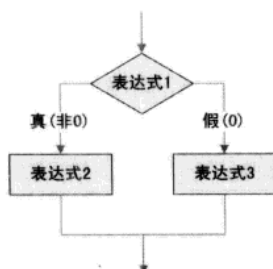


图4.11 条件表达式流程图

可以看出，条件表达式语句等同于下面形式的 if 语句：

```
if(表达式 1)
    表达式 2;
else
    表达式 3;
```

条件表达式通常用于赋值语句之中。例如，上面求两个数中最大数的程序中的 if-else 语句部分，可用条件表达式写为：

```
max=(a>b)?a:b;
```

执行该语句的语义是：如果 a>b 为真，则把 a 赋予 max，否则把 b 赋予 max。

4.3.7 应用条件运算符注意事项

应用条件运算符时需注意以下几点。

- ☑ 条件运算符的运算优先级低于关系运算符和算术运算符，但高于赋值运算符。

```
max=(a>b)?a:b
```

可以去掉括号写为：

```
max=a>b?a:b
```

但是，这样会给阅读程序带来一些不便，不建议这样编写程序。

- ☑ 在条件运算符中，“?”和“:”是一对运算符，不能分开单独使用。例如，下面的表达式都是错误的。

```
max=(a>b)?a;
max=a:b;
```

- ☑ 条件运算符的结合方向是自右至左的。例如：

```
a>b?a:c>d?c:d
```

应理解为:

```
a>b?a: (c>d?c:d)
```

☐ 条件表达式可以嵌套使用,也就是说,表达式2和表达式3又可以是一个条件表达式。这样就可以实现很多复杂的分支结构。例如:

```
a>b?a:c>d?c:d>e?d:e;
```

虽然条件表达式可以嵌套使用实现许多复杂的分支结构,但是不建议多次嵌套使用条件表达式编写复杂的程序,因为这样的程序阅读起来不容易理解,易读性差。

4.3.8 switch 语句的解释

在开始介绍 switch 语句之前,先来看一个划分年龄称谓的程序。我们伟大的祖先曾经使用了一些经典的对于不同年龄人的称谓,比如“而立”、“花甲”之类,而对于现在的人来说,这些词语已经渐渐地被遗忘了,下面的程序就来带你捡起这些快被遗忘的经典年龄称谓:

```
#include <stdio.h>

void main()
{
    int age;
    printf("\n 请输入你要查询的年龄: (年龄包括 30、40、50、60、70) \n");
    scanf("%d",&age);      //输入年龄
    if(age==30)             //年龄为 30 的情况
    {
        printf("\n 古人将 30 岁叫做而立");
    }
    else if(age==40)        //年龄为 40 的情况
    {
        printf("\n 古人将 40 岁叫做不惑");
    }
    else if(age==50)        //年龄为 50 的情况
    {
        printf("\n 古人将 50 岁叫做知天命");
    }
    else if(age==60)        //年龄为 60 的情况
    {
        printf("\n 古人将 60 岁叫做花甲");
    }
    else if(age==70)        //年龄为 70 的情况
    {
        printf("\n 古人将 70 岁叫做古稀");
    }
}
```

上面的程序有一个特点,那就是条件都是对变量 age 进行判断的,也就是说所有的 if 语句都是在进行同一类型的判断,编写这么多的 if 语句既使程序显得复杂,又使程序的可读性差。针对这种情况,C 语言还提供了另一种用于多分支选择的语句——switch 语句。

switch 语句又称开关语句,其一般形式为:



```
switch(表达式)
{
    case 常量表达式 1: 语句 1;
    case 常量表达式 2: 语句 2;
    ...
    case 常量表达式 n: 语句 n;
    default:           语句 n+1;
}
```


其语义是：计算表达式的值。并逐个与其后的常量表达式的值相比较，当表达式的值与某个常量表达式的值相等时，即执行其后的语句，不需要再进行判断，直接执行后面所有 case 后的语句。如表达式的值与所有 case 后的常量表达式均不相同时，则执行 default 后的语句。

其中 default 分支是可选的。如果没有 default 分支也没有其他分支与表达式的值相等，则该 switch 语句不执行任何动作。各个 case 分支和 default 分支的排列次序是任意的。此处用 switch 语句来实现划分年龄称谓的程序，程序清单如下所示：

```
#include <stdio.h>

void main()
{
    int age;
    printf("\n 请输入你要查询的年龄：（年龄包括 30、40、50、60、70）\n");
    scanf("%d",&age);      //输入年龄
    switch(age)
    {
        case 30: printf("\n 古人将 30 岁叫做而立");    break; //年龄为 30 的情况
        case 40: printf("\n 古人将 40 岁叫做不惑");    break; //年龄为 40 的情况
        case 50: printf("\n 古人将 50 岁叫做知天命");    break; //年龄为 50 的情况
        case 60: printf("\n 古人将 60 岁叫做花甲");    break; //年龄为 60 的情况
        case 70: printf("\n 古人将 70 岁叫做古稀");    break; //年龄为 70 的情况
        default: printf("\n 本程序无法查询到你输入的年龄的称谓");
    }
}
```

此程序首先要求输入一个要查询的年龄并存放在变量 age 中，然后根据 age 的值和各个 case 中的常量表达式进行匹配，匹配上后就会执行 case 块中的输出语句，最后执行 break 语句，跳出 switch 语句。

 **说明：**此程序中出现了 break 语句，其输入后面要介绍的跳转语句中的一种。此处 break 的作用是跳到 switch 语句之外，而不继续执行下面的 case 语句块。假设输入的年龄值为 30，那么将执行 case 30 语句块，输出后会执行其后的 break 语句，这时，会跳出 switch 语句，而不会执行后面的 case 40 以及其后的所有 case 语句块。

可以看出，使用 switch 语句编写的程序既简洁易读，又可以提高运行效率。所以，在程序需要对一个变量进行多种条件的判断时，应该使用 switch 语句编写程序。

4.3.9 应用 switch 语句注意事项

应用 switch 语句时需要注意以下几点。

- ❑ 在 case 后的各常量表达式的值不能相同，如果相同，就会出现对于表达式的同一个值，有两种或者多种执行方案，这时程序就会出现错误。
- ❑ 在 case 后，允许有多个语句，可以不用“{}”括起来。例如下面的语句是允许的：

```
case:a=0;  
    b=1;  
    c=2;
```

- ❑ 多个 case 可以共享一组执行语句，例如下面的语句是允许的：

```
case 10:  
case 9: grade='A'; break;
```

表示表达式的值等于 10 或者 9 都执行语句 grade='A'。

4.3.10 分支结构程序设计方法

前面的小节已经介绍了 3 种分支结构的实现方式，即 if 语句、条件运算符、switch 语句。现在，当一个存在着分支结构的问题出现后，要如何来分析问题并使用这些分支结构的实现方式组成一个完整且有用的程序呢？

分支结构程序设计的方法有很多种，此处介绍一种简单有效的方法。

第 1 步 确定分支条件。

第 2 步 按照题目要求画出程序的流程图，此处的流程图可以不是整个程序的流程图，而只是整个程序的流程图中与分支结构相关的部分。还是要提醒的是，不管流程图有多简单，养成画流程图的习惯绝对有好处。

第 3 步 根据前面的流程图选择一种或几种分支结构的实现方式。

第 4 步 按照选择的实现方式和流程图写出若干个程序块。

第 5 步 将所有的程序块按照逻辑进行组合，并加上需要包含的头文件等其他内容，形成最终的程序。

现在按照上面的方法，来编程解决如下问题：编程实现接收用户输入年份，并判断此年份是否是闰年。

【分析】闰年的判断规则是：如果年份能被 4 整除，且不能被 100 整除，是闰年；能被 400 整除，也是闰年。

第 1 步 确定分支条件。根据题目中给出的规则，可以确定以下 3 个分支条件：

- ❑ 年份是否可以被 4 整除。
- ❑ 年份是否可以被 100 整除。
- ❑ 年份是否可以被 400 整除。

第 2 步 画程序流程图。根据上一步确定的分支条件和题目的要求，画出这个题目的流程图（只是分支结构部分的流程图），如图 4.12 所示。当然，正确的流程图并非只有下面这一种。

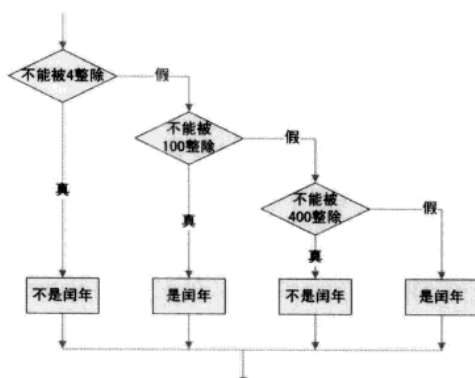


图4.12 程序流程图

第3步 选择实现方式。从流程图可以看出，采用 `else if` 形式的 `if` 语句来实现是一个较好的方式。

第4步 写出语句块。假设定义一个变量 `sign` 作为是否为闰年的标志，定义一个变量 `year` 作为年份。则按照流程图和前面确定的实现方式，写出以下程序块。

☑ 程序块一：

```
if(year%4!=0)
{
    sign=0;
}
```

☑ 程序块二：

```
else if(year%100!=0)
{
    sign=1;
}
```

☑ 程序块三：

```
else if(year%400!=0)
{
    sign=0;
}
```

☑ 程序块四：

```
else
{
    sign=1;
}
```

第5步 编程实现。有了上一步的各个分支的程序块，现在只需要定义一些变量，以及按要求增加一些输入、输出，并加上一些头文件和其他文件就可以编写出完整的程序了。判断年份是否是闰年的程序清单如下所示：

```
#include <stdio.h>

void main()
```

```
{
    int year, sign;        //定义表示年份和闰年标志的变量
    printf("请输入需要判别的年份: \n");
    scanf("%d", &year); //输入年份
    if(year%4!=0)
    {
        sign=0;
    }
    else if(year%100!=0)
    {
        sign=1;
    }
    else if(year%400!=0)
    {
        sign=0;
    }
    else
    {
        sign=1;
    }
    if(sign)
    {
        printf("%d 年是闰年", year);
    }
    else
    {
        printf("%d 不是闰年", year);
    }
}
```

因为按照题目的要求程序应该能够输出年份是否是闰年，所以此程序的最后，用一个 if...else 形式的 if 语句对闰年标识 sign 进行了判断，如果表示为真（非 0），则输出年份是闰年；如果为假（0），则输出年份不是闰年。这其实又是一个分支结构，可以再分 5 个步骤进行分析和实现，因为这个逻辑相对简单，此处对这个步骤不再介绍。

对此程序，因为各人对分支的理解、流程图的画法，以及对各种实现分支结构的方式的偏好都有所不同，所以最终实现的程序也不尽相同。例如，下面的程序段就是对这个例子最简洁的实现方式之一：

```
if((year%4==0&&year%100!=0)|| (year%400==0))
{
    printf("%d 年是闰年", year);
}
else
{
    printf("%d 年不是闰年", year);
}
```



4.4 循环结构

如果程序中出现了要多次重复执行的语句时，顺序结构和分支结构就无法对其进行处理，从而引入了循环结构。C 语言中提供了 3 种实现循环结构的语句：**while** 语句、**do-while** 语句、**for** 语句。本节将对这些语句进行详细介绍，并结合实例介绍一种实现循环结构的方法。

4.4.1 什么是循环结构

循环结构就是在满足某个条件之前反复执行一个语句序列。这个语句序列叫做循环体，如图 4.13 所示。

要理解循环结构，还是看一下句子组合练习。不过，此处的组合练习存在一些做重复事情的分句，我们需要将这些分句总结和提炼，最终组成一个简练并能够总结所有分句意思的句子，例如：

- ☑ 新影片叫《阿甘正传》。
- ☑ 电影院周一在放映新影片。
- ☑ 电影院周二在放映新影片。
- ☑ 电影院周三在放映新影片。

来分析一下这 4 个分句。第一个分句说明了新影片的名字，后面的 3 个分句其实说的都是一件事：电影院放映新影片。只不过 3 个分句中的时间不同。不难看出，组合的句子应该为：周一到周三这段时间，电影院都在放映新影片《阿甘正传》。这个句子中周一到周三是放映影片的时间段，也可以看做循环地做放映新影片这件事的条件；放映新影片是具体的事情，也就是循环的主体，即循环体。

在具体实现方式上，C 语言循环结构中循环过程是通过某种循环语句来实现的，循环条件一般就是前面章节学过的逻辑表达式或条件表达式，循环体可以是单条语句，也可以是复合语句。

4.4.2 关于 while 语句的解释

while 循环是使用入口条件的有条件循环。**while** 语句的一般形式为：

```
while(表达式) 语句
```

其中表达式是循环条件，语句为循环体。

while 语句的语义是：计算表达式的值，当值为真（非 0）时，执行循环体语句；当值为假（0）时，跳出循环。其执行过程如图 4.14 所示。

在构造 **while** 循环时，有一点至关重要：当构造一个 **while** 循环时，循环中必须包含能改变判断表达式的值使表达式的值最终变成假（0）。否则，循环将永远执行下去，形成所谓的死循环，例如下面的程序段：

```
int i=1;
```

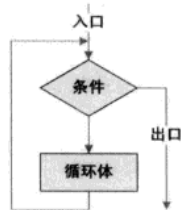


图4.13 循环结构

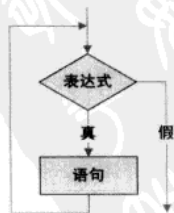


图4.14 while语句执行过程

```
while(i<10)
    printf("在执行循环");
```

因为 i 的值始终是 1, 也就是说, 永远满足循环条件 $i < 10$, 所以, 程序将不断地输出“在执行循环”, 永远不会终止。要使程序终止, 可将程序改写为:

```
int i=1;
while(++i<10)
    printf("在执行循环");
```

这样, 程序每执行一次循环, 就对 i 加 1, 运行 8 次以后, 循环条件为假, 循环终止。

while 循环被称为有条件循环是因为语句部分的执行要依赖于判断表达式中的条件。说使用入口条件是因为在进入循环体之前必须满足这个条件。如果在第一次进入循环体的时候条件就没有被满足, 程序将永远不会进入循环体。例如:

```
int i=11;
while(i<10)
    printf("在执行循环");
```

因为 i 一开始就被赋值为 11, 不符合循环条件 $i < 10$, 所以不会执行后面的输出语句。要使程序能够进入循环, 必须给 i 赋比 10 小的初值。

4.4.3 应用 while 语句注意事项

应用 **while** 语句需要注意以下几点。

- **while** 语句中的表达式一般是关系表达式或逻辑表达式, 但是在语法上所有的表达式都可以作为循环条件。只要表达式的值为真 (非 0) 即可继续循环。例如下面的程序段:

```
int x=10;
while(y=x--)
    printf("%d",y);
```

在这段程序中, **while** 循环的表达式是一个赋值语句, 当 $x--$ 的值不是 0 的时候, 循环表达式的值为真, 执行循环; 当 $x--$ 经过几次运算, 值为 0 时, 循环表达式的值为假, 循环终止。

- 只有位于判断条件之后的单个语句 (简单语句或者复合语句) 才是循环体。缩进只是为了帮助阅读程序, 而不会改变程序的执行逻辑。例如下面的程序段:

```
int x=1;
while(x<10)
    printf("%d",x);
    x++;
```

在这段程序的中, 循环条件为 $x < 10$, 而循环体仅仅只有语句 `printf("%d",x);`, 所以 x 的值将永远小于 10, 循环无法终止, 形成死循环。要解决这个问题, 有一个简单有效的办法, 就是不管需要的循环体是简单语句还是复合语句, 都用 “{}” 将循环体括起来。这样既防止出错, 又容易阅读。上面的程序用此方法修改后为:

```
int x=1;
while(x<10)
{
    printf("%d",x);
    x++;
}
```



4.4.4 关于 do...while 语句的解释

在开始介绍 do...while 语句之前, 先来看一个用 while 语句实现的程序, 程序清单如下所示:

```
#include <stdio.h>

void main()
{
    int i;
    scanf("%d",&i);
    while(i>100)           //循环条件
        printf("i 的值大于 100\n"); //循环体
}
```

正如在介绍 while 语句时讲到的, while 语句是使用入口条件的, 在进入循环体之前必须满足这个条件。如果在第一次进入循环体的时候条件就没有被满足, 那么程序将永远不会进入循环体。在上面的程序中, 是否进入循环完全取决于用户的输入, 如果用户输入的值小于 100, 那么程序将无法进入循环体。

如果现在有这样一种需求: 不管循环的条件是什么, 都要保证至少进行一次循环。while 语句明显不能满足这种需求, 而 C 语言中提供了一种能满足这种需求的语句——do...while 语句。do...while 循环叫做退出条件循环, 判断条件在执行循环之后进行检查, 这样就可以保证循环体至少被执行一次。do-while 语句的一般形式为:

```
do
    语句
while(表达式);
```

此循环与 while 循环的不同在于: do...while 循环先执行循环中的语句, 再判断表达式是否为真, 如果为真, 则继续循环; 如果为假, 则终止循环。因此, do...while 循环至少要执行一次循环语句。语句部分可以是简单语句也可以是复合语句。do...while 本身是一个语句, 所以需要以一个分号结束, 执行过程如图 4.15 所示。

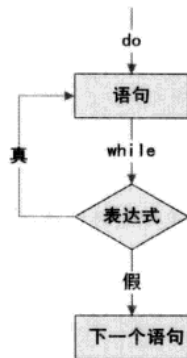


图4.15 do...while语句执行过程图

4.4.5 应用 do...while 语句时防止死循环

应用 do...while 循环特别要防止产生死循环。除了前面讲到的引起 while 循环产生死循环的情况外,如果在 do...while 循环中,do 后面的语句和循环条件相悖,也会出现死循环,例如下面的程序段:

```
int i=1;
do
    i--;
while(i++<10);
```

这段程序将会出现死循环。程序首先执行 do 后面的语句,因为变量 i 的初值为 1,所以执行语句 i-- 后 i 的值变为 0,然后执行循环条件 i++<10,因为满足循环条件所以继续执行循环体。虽然在循环条件中 i 的值每次会加 1,但是在循环体中 i 的值每次会减 1。这样, i 的值将永远满足小于 10 的条件,循环无法终止,从而产生了死循环。

有以下两种方法可以防止这种错误。

- ❑ 必须保证循环条件和循环体要同步地促使循环终止条件的产生,也就是说,要么同时增,要么同时减。像上面的程序中,要使 i 的值小于 10,必须让 i 的值在执行完循环后有增长,增长可以在循环体中,也可以在循环条件中,或者两者中同时出现。
- ❑ 促使循环终止条件的速度必须要比阻止循环终止条件的速度快。也可以理解为促使循环终止条件的步伐要比阻止循环终止条件的步伐大,这有点类似于人们常说的走两步退一步。例如上面的程序可以改为:

```
int i=1;
do
    i--;
while(i+2<10);
```

4.4.6 不确定循环和计数循环

前面学过的 while 语句或 do...while 语句构成的循环有些是不确定循环。也就是说,在表达式变为假之前不能预知循环要执行多少次。例如下面的程序段:

```
while((c=getchar())!='\n')
    putchar(c);
```

这段程序的作用是:当用户输入的不是回车字符时,输出用户输入的字符并等待用户再次输入字符;当输入的是回车字符时,循环终止。而用户何时输入回车字符是未知的,所以循环执行的次数不确定,属于不确定循环。

而有些循环执行的次数是预先能够确定的,将这种循环称为计数循环。计数循环程序清单如下所示:

```
#include <stdio.h>

void main()
{
    int end=100;
    int count=1;           //初始化
    while(count<=end)      //循环条件
```



```
{  
    printf("%d\t",count); //动作  
    count++;             //更新计数  
}
```

这是一个很简单的程序，其作用是输出 1~100 之间的数，可以看出，这个循环的循环体将执行 100 次。

尽管上面的程序能够很好地工作，但它并不是这种情况下的最好选择，因为定义循环的动作没有被有效地组织起来。下面来详细介绍这一点。

在建立一个重复执行固定次数的循环时，要设计 3 个动作：

- ☑ 必须初始化一个计数器。
- ☑ 计数器与某个有限的值进行比较。
- ☑ 每次执行循环，计数器的值都要更新（递增或者递减）。

在上面的程序中，while 循环条件执行比较的动作，递增运算符“++”执行更新的动作。更新在循环体的最后一行处执行。这确实不是一个好的选择，因为一旦遗漏了这个更新动作，程序将形成死循环。所以更好的办法是使用 `count++<=end` 来将判断和更新操作放在一个表达式中。但即使这样还存在一个问题，计数器的初始化仍然在循环体外执行，这样就有可能忘记初始化。其实就算是一些资深的程序员，这种事情也会经常发生。所以需要一种可以避免这些问题的循环语句，在 C 语言中，此循环语句就是 for 语句。

4.4.7 关于 for 语句的解释

在 C 语言中，提供了一种能将上面所说的 3 个动作（初始化、判断、更新）放在一起的语句——for 语句。其一般形式为：

for(表达式 1;表达式 2;表达式 3) 语句

执行过程如下。

第 1 步 求解表达式 1。

第 2 步 求解表达式 2，若其值为真（非 0），则执行 for 语句中指定的内嵌语句，然后执行下面的第 3 步；若其值为假（0），则终止循环，转到第 5 步。

第 3 步 求解表达式 3。

第 4 步 转回上面第 2 步继续执行。

第 5 步 循环结束，执行 for 语句下面的一个语句。

这个执行过程如图 4.16 所示。

for 语句最简单的应用形式也是最常用的形式如下：

for(循环变量赋初值;循环条件;循环变量更新) 语句

- ☑ 循环变量赋初值总是一个赋值语句，是用来给循环控制变量赋初值。在循环开始执行时执行，并且只执行一次。
- ☑ 循环条件一般是一个关系表达式，决定什么时候退出循环。

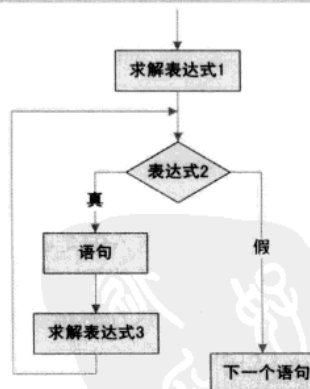


图4.16 for语句执行过程图

- ☐ 循环变量更新，定义循环控制变量每循环一次后，按什么方式变化。一般情况下是一个递增或者递减的语句。

这3个部分之间用“;”分开。例如：

```
for(i=1; i<100; i++)
    sum+= i;
```

这个 for 循环先给变量 i 赋初值 1，然后判断 i 是否小于 100，若为真则执行语句，执行完语句后 i 的值增加 1。再重新判断，直到条件为假，即 $i \geq 100$ 时，终止循环。

用 while 语句实现为：

```
i=1;
while (i<100)
{
    sum +=i;
    i++;
}
```

可以看出，这种形式的 for 循环，就是如下形式的 while 循环：

```
表达式 1;
while (表达式 2)
{
    语句
    表达式 3;
}
```

用 for 语句来实现上面的计数循环，程序清单如下所示：

```
#include <stdio.h>

void main()
{
    int end=100;
    for(count=1;count<=end; count++)    //for 循环
    {
        printf("%d\t",count);          //动作
    }
}
```

比较两个程序，可以看出：用 for 语句来实现计数循环时，赋初值、判断和更新 3 个动作以一个整体的形式呈现，程序简洁，不易出错。所以在编写计数循环的程序时，最好也是最常用的形式是使用 for 语句。

4.4.8 使用 for 语句注意事项

for 循环中的“表达式 1（循环变量赋初值）”、“表达式 2（循环条件）”和“表达式 3（循环变量更新）”都是可选择项，即可以省略，但“;”不能省略。

如果省略了“表达式 1（循环变量赋初值）”，表示不对循环控制变量赋初值。如果省略了“表达式 2（循环条件）”，则不做其他处理时便成为死循环。例如：

```
for(i=1;;i++)sum=sum+i;
```



如果省略了“表达式 3 (循环变量更新)”，则不对循环控制变量进行更新操作，这时可在语句体中加入修改循环控制变量的语句。例如：

```
for(i=1;i<=100;)
{
    sum=sum+i;
    i++;          //对循环变量更新
}
```

甚至，3 个表达式都可以省略。例如：

```
for(;;)语句
```

相当于：

```
while(1)语句
```

可以用减运算符来减少计数器而不是增加它。例如：

```
for(i=10;i>1;i--)
    sum+=i;
```

如果需要的话，可以让计数器依次加或者减 2、3...例如：

```
for(i=1;i<100;i+=5)
    sum+=i;
```

可以让数量几何增长，而不是算术增长。所谓几何增长，就是说不是每次加一个固定数，而是乘上一个固定数。例如：

```
for(i=10.0;i<100.0;i*=1.5)
    sum+=i;
```

用来计数的可以不是数字，而是字符。例如：

```
for(c='a';c<='z';c++)
    printf("%c\t",c);
```

表达式 1 可以是设置循环变量的初值的赋值表达式，也可以是其他表达式。例如：

```
int i=1;
for(printf("这个循环计算 1 到 100 之间的值: ");i<=100;i++)
    sum=sum+i;
```

表达式 1 和表达式 3 可以是一个简单表达式也可以是逗号表达式。例如：

```
for(sum=0,i=1;i<=100;i++)
    sum=sum+i;
```

或：

```
for(i=0,j=100;i<=100;i++,j--)
    sum=i+j;
```

表达式 2 一般是关系表达式或逻辑表达式，但也可以是数值表达式或字符表达式，只要其值非零，就执行循环体。例如：

```
for(i=0;(c=getchar())!='\n';i+=c);
```

循环中的动作可以改变循环表达式的参数。例如：

```
for(i=1;i<100;i=i+step)
```

此处的 `step` 是一个可变的参数, 假设执行几次循环之后, 发现 `step` 的值太小或者太大, 可以在循环中使用 `if` 语句来改变 `step` 的大小, 如下所示:

```
for(i=1, step=1; i<100; i=i+step)
{
    sum+=i;
    if(i>10)
        scanf("%d", &step);
}
```

此处要提醒一点, 在上面的程序中, `step` 的值可由用户在循环体内指定, 这是很危险的。如果将 `step` 指定为 0, 循环将无法终止, 形成死循环。

4.4.9 选择哪种循环实现方式

前面介绍过了 3 种实现循环的语句: `while` 语句、`do...while` 语句、`for` 语句。那么, 当确定要使用循环的话, 应该选择哪种循环呢? 规则是: 首先要判断需要的是入口条件循环还是退出条件循环, 如果是退出条件循环, 最好使用 `do...while` 语句; 而如果是入口条件循环, 则应该使用 `for` 语句或者 `while` 语句的一种。这时候还要看是不确定循环还是计数循环, 如果是不确定循环, 最好选用 `while` 循环; 而如果是计数循环, 最好是选用 `for` 循环。

选择循环的规则并不是很难, 但是对一些人来说记忆这些规则显得有些麻烦。其实, 对于一个学习完 C 语言分支结构的人来说, 完全可以编写一个程序去实现对使用哪种循环语句的判断。选择使用哪种循环的程序清单如下所示:

```
#include <stdio.h>

void main()
{
    char answer;
    printf("\n你所要做的循环是退出条件循环吗(是请选择 y, 否请选择 n)? \n");
    answer=getchar();
    if(answer == 'y')
    {
        printf("\n你应该使用 do...while 循环\n");
    }
    else
    {
        printf("\n你所要做的循环是计数循环吗(是请选择 y, 否请选择 n)? \n");
        answer=getchar();
        if(answer == 'y')
        {
            printf("\n你应该使用 for 循环\n");
        }
        else
        {
            printf("\n你应该使用 while 循环\n");
        }
    }
}
```



上面的程序有一个缺陷，那就是当用户输入字符'y'和'n'以外的其他字符时，程序的处理方式和输入了'n'一致。有兴趣的读者可以修改这个程序。这里就不再赘述了。

4.4.10 循环中的循环

所谓循环中的循环，即嵌套循环，是指既是一个循环，同时又是另一个循环的循环体。内嵌的循环中还可以嵌套循环，就是多层嵌套循环。

循环有 3 种实现方式：

- ☑ while 循环。
- ☑ do...while 循环。
- ☑ for 循环都可以互相嵌套。

如果是两层循环，就可以组合成 9 种形式的嵌套循环。最常用的嵌套循环是两层 for 循环。这种结构中，一个循环处理一行中所有的列，而另外一个循环处理所有的行。例如一个输出 5 行 10 列 “*” 的程序，程序清单如下所示：

```
#include <stdio.h>

void main()
{
    int i,j;                //定义用作计数器的变量 i 和 j
    for(i=1;i<=5;i++)       //外部循环
    {
        for(j=1;j<=10;j++) //内部循环
        {
            printf("*");    //输出*
        }
        printf("\n");       //换行
    }
}
```

本程序在 Visual C++ 6.0 环境下运行结果如图 4.17 所示。

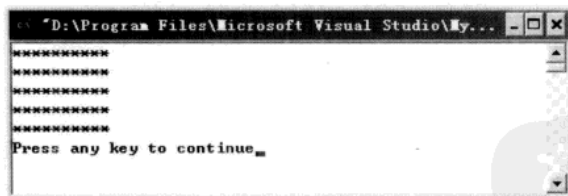


图4.17 输出结果图

在上面的程序中，第 6 行的 for 循环被称之为外部循环，而第 8 行的 for 循环被称之为内部循环。可以看出，外部循环控制的是行，内部循环控制的是列。程序在进入外部循环的循环体后，先执行内部循环；执行完一次内部循环后，再执行第 12 行的语句进行换行；然后更新外部循环的计数器，如果外部循环的条件仍然满足，再次进入外部循环的循环体，执行内部循环和其后的换行语句，直到外部循环的条件不满足，终止整个循环。

从上面的程序中可以看出,内部循环在外部循环的每个周期中做同样的事情。通过使内部循环的一部分依赖于外部循环,可以使内部循环在每个周期中表现不同。例如,下面的程序对输出 5 行 10 列 “*” 的程序做了少许的修改,使内部循环依赖于外部循环,从而产生了不同的输出。程序清单如下所示:

```
#include <stdio.h>

void main()
{
    int i,j;                //定义用作计数器的变量 i 和 j
    for(i=1;i<=5;i++)       //外部循环
    {
        for(j=1;j<=i;j++)   //内部循环
        {
            printf("*");    //输出*
        }
        printf("\n");       //换行
    }
}
```

本程序在 Visual C++ 6.0 环境下运行结果如图 4.18 所示。

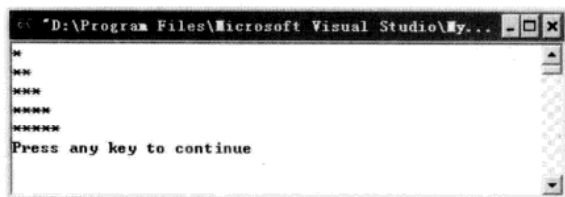


图4.18 输出结果图

此程序只对输出 5 行 10 列 “*” 的程序修改了一处,就是内部循环的循环条件由 $j \leq 10$ 变成了 $j \leq i$ 。因为 i 是外部循环的计数器,所以每次进入外部循环的循环体执行内部循环时, i 的值都不相同,从而使每次执行内部循环的次数也不相同。所以输出的结果有 5 行,是外部循环的循环次数,而每一个行的列数是每次执行内部循环的次数,正好和所处的行号相同。

4.4.11 循环结构程序设计方法

前面小节介绍了 3 种循环结构的实现方式: while 语句、do...while 语句、for 语句。同时介绍了嵌套循环。现在,当一个存在着循环结构的问题出现时,要如何来分析问题并使用这些循环结构的实现方式组成一个完整且有用的程序呢?

设计循环结构的程序的方法有很多种,此处介绍一种简单有效的方法。

第 1 步 根据题目要求和选择循环的原则选择一种或几种循环结构的实现方式。

第 2 步 确定每个循环的终止条件,初始化语句和更新语句。

第 3 步 确定每个循环的循环体。确定循环体的顺序是由最内层循环开始,到最外层循环结束。



第4步 画出程序的流程图。

第5步 按照流程图将所有的程序块进行组合，并加上需要包含的头文件等其他内容，形成最终的程序。

按照上面的方法，来看一个中国古代数学家张丘建在他的《算经》中提到的著名的“百钱买百鸡问题”：鸡翁（公鸡）一值钱五，鸡母（母鸡）一值钱三，鸡雏（小鸡）三值钱一。百钱买百鸡，问鸡翁、鸡母、鸡雏各几只？如图 4.19 所示。

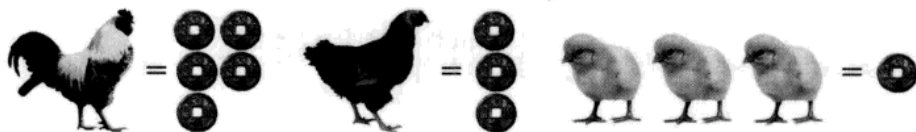


图4.19 百钱买百鸡

【分析】假设鸡翁、鸡母、鸡雏的个数分别为 x 、 y 、 z ，根据题意给定用百钱买百鸡，若全买公鸡最多买 20 只，显然 x 的值在 0~20 之间；同理， y 的取值范围在 0~33 之间，可得到下面的不定方程：

$$5x + 3y + z/3 = 100$$

$$x + y + z = 100$$

所以此问题就变为求这个不定方程的整数解的问题。和数学上求不定方程解不同，由程序设计实现不定方程的求解在分析确定方程中未知数变化范围的前提下，可通过对未知数可变范围的穷举，验证方程在什么情况下成立，从而得到相应的解。

第1步 确定实现方式。

根据题意，此程序应该包括两层循环，一层循环控制公鸡的数量，一层循环控制母鸡的数量，所以选用两层嵌套的 for 循环实现比较合理。此处，笔者用外层循环控制公鸡数量，内层循环控制母鸡数量。

第2步 确定每个循环的终止条件，初始化语句和更新语句。

先确定内层循环。因为内层循环控制的是母鸡数量，其数量是从 0 递增到 33。所以，假设内层循环的计数变量为 y ，则循环的终止条件为 $y \leq 33$ ，初始化语句为 $y=0$ ，更新操作需要 y 的值每次递增 1，语句为 $y++$ ；。

再确定外层循环。因为外层循环控制的是公鸡数量，其数量是从 0 递增到 20。所以，假设外层循环的计数变量为 x ，则循环的终止条件为 $x \leq 20$ ，初始化语句为 $x=0$ ，更新操作需要 x 的值每次递增 1，语句为 $x++$ ；。

第3步 确定每个循环的循环体。

先确定内层循环的循环体。每次进入内层循环后，因为已经有了公鸡和母鸡的数量，所以首先要确定小鸡的数量，一共有 100 只鸡，假设小鸡的数量是 z ，则 $z=100-x-y$ 。接下来，因为 x 、 y 、 z 的数量满足条件 $(5*x+3*y+z/3==100)$ ，并且 z 的数量必须是整数，也就是说， $z\%3==0$ ，这些都是判断的条件，可用一个 if 语句来表示，而 if 语句的作用就是输出各种鸡的数量。

再确定外层循环的循环体。因为外层循环只是控制公鸡的数量，所以其循环体只包括内层循环。

第4步 画程序的流程图。

根据题目的要求，画出这个题目的流程图如图 4.20 所示（只画出循环部分）。当然，正确的流程图并非只有下面这一种。

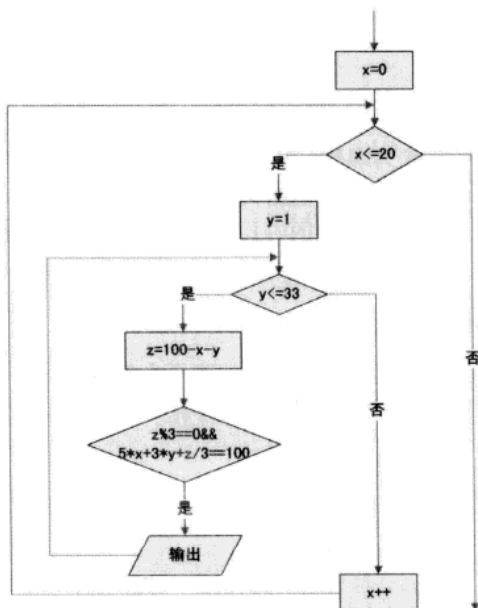


图4.20 程序流程图

第5步 编程实现。

有了程序的流程图和确定的内外层循环，现在只需要定义一些变量，并加上一些头文件和其他文件就可以编写出完整的程序了。“百钱买百鸡问题”的程序清单如下所示：

```

#include<stdio.h>

void main()
{
    int x,y,z;           //各种鸡的数目
    int i=0;             //方程解数目

    printf("百钱买百鸡，求鸡翁、鸡母、鸡雏的数目：\n");

    for(x=0;x<=20;x++)   //外层循环控制鸡翁数
    {
        for(y=0;y<=33;y++) //内层循环控制鸡母数
        {
            z=100-x-y;    //鸡雏数
            if(z%3==0&&5*x+3*y+z/3==100)

```




```
{  
    //验证取 z 值的合理性及得到一组解的合理性  
    printf("%2d: 鸡翁数: %2d, 鸡母数: %2d, 鸡雏数: %2d\n", ++i, x, y, z);  
}  
}
```

4.5 跳转结构

C 语言中除了提供了上面讲到的 3 种基本结构外,还提供了一种可以使程序在执行过程中跳转到指定位置的结构,称之为跳转结构。跳转结构一般在程序中与循环结构或者分支结构配合使用,所以也可以认为它是为其他结构来服务的。

C 语言中提供了 3 种实现跳转结构的语句: `break` 语句、`continue` 语句和 `goto` 语句。本节将对其进行详细介绍,以及介绍一些使用到跳转结构的程序。

4.5.1 什么是跳转结构

跳转结构是指在碰到跳转语句时,程序不按照原来的顺序执行,而是根据跳转语句,跳出或者转到程序的某个位置执行。假设原来的程序是一个循环结构,在加入跳转语句后,如图 4.21 所示。

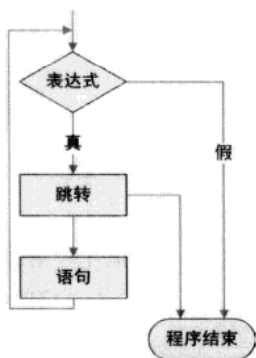


图4.21 跳转结构

要理解跳转结构,还是看一下句子组合练习。不过,此处的组合练习中,存在一个按照前面讲到的 3 种基本流程组合好的句子和另外的一些分句,例如:

- ☑ 小明每天 7:30 分乘坐公交车去上学。
- ☑ 因为出现交通事故,昨天公交车到达小明要乘车的车站推迟了 20 分钟。
- ☑ 昨天小明在 7:40 分乘坐出租车去上学。

来分析一下这 3 个分句。第一个分句和第三个分句都说明了一件事:小明乘坐某种交通工具去上学。第一个分句可以看成前面讲到的循环结构的句子,是一种正常情况。而第二个分句是一种特殊情况,因为这种特殊情况,导致产生了第三个分句。组合后的句子为:通常

情况下, 小明每天 7:30 分乘坐公交车去上学; 昨天, 因为出现交通事故, 公交车到达小明要乘车的车站推迟了 20 分钟, 所以小明在 7:40 分乘坐出租车去上学, 如图 4.22 所示。

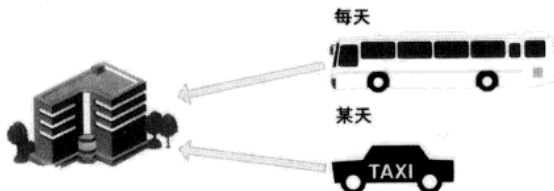


图4.22 不同交通工具上学

如果把去上学看成是整个事件的结尾, 那么这个事件有两种类型的结尾。同样, 存在跳转结构的程序可能会存在多个入口条件和出口条件, 影响程序的清晰度和易读性, 所以要慎用跳转结构。

4.5.2 break 语句

break 语句的一般形式为:

```
break;
```

在介绍 switch 语句时已经介绍过 break 语句, 它的作用是使程序跳出 switch 语句而执行其后的语句。break 语句用于 switch 语句的一般形式为:

```
switch(表达式)
{
    case 常量表达式 1: 语句 1; break;
    case 常量表达式 2: 语句 2; break;
    ...
    case 常量表达式 n: 语句 n; break;
    default:           语句 n+1; [break];
}
```

其中“[]”表示可选。break 语句用于 switch 语句中的执行过程如图 4.23 所示。

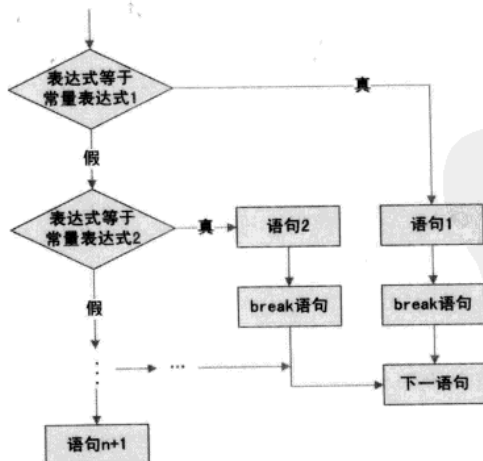


图4.23 break语句用于switch语句中的执行过程



`break` 语句除了能用于 `switch` 语句中外,还可以用于 `do...while`、`for`、`while` 循环语句中,使程序终止循环而执行循环后面的语句。通常情况下,当循环体内出现了某种特殊情况时才要求跳出循环,所以 `break` 语句总是与 `if` 语句连在一起。即满足条件时便跳出循环。

假设 `break` 语句用于 `while` 循环语句中,要求在某个条件下跳出循环,其一般形式是:

```
while(表达式1)
{
    ...
    if(表达式2) break;
    ...
}
```

这种形式的流程图如图 4.24 所示。

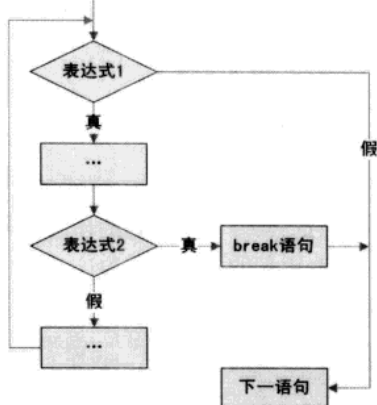



图4.24 `break`语句用于`while`语句中的执行过程

下面是一个用来判断用户输入的数 `n` 是否为素数的程序。程序清单如下所示:

```
#include <stdio.h>

void main()
{
    int i,n;
    printf("请输入一个整数\n");
    scanf("%d",&n);
    i=2;
    while(i<n)
    {
        if(n%i==0)
            break;      //跳出循环
        i++;
    }
    if(i==n)
        printf("%d 是素数\n",n);
    else
        printf("%d 不是素数\n",n);
}
```

这个程序首先接收用户输入的数 n ，循环变量 i 初值为 2。在循环体中，先用 i 对 n 求余，如果求余结果是 0，则说明 n 不是素数，使用 `break` 语句跳出循环。因为循环是在循环条件 ($i < n$) 被满足的情况下终止的，所以程序会执行第 18 行的 `printf` 语句，输出用户输入的数不是素数。如果求余结果是 0 的条件在每次进入循环体后都不满足的话，则每次都会执行第 12 行的语句，执行若干次后，循环条件 ($i < n$) 不满足，循环终止，这时 i 的值和 n 的值相等，程序执行第 15 行的 `printf` 语句，输出用户输入的数是素数。

 注意：`break` 语句不能用于循环语句和 `switch` 语句外的任何语句中，并且，在多层循环中，一个 `break` 语句只向外跳一层。

4.5.3 continue 语句

`continue` 语句的一般形式为：

```
continue;
```

`continue` 语句用在 `for`、`while`、`do-while` 循环体中，作用是终止本次循环，就是跳过循环体中尚未执行的语句，接着进行下一次是否执行循环的判断。

通常情况下，`continue` 语句总是与 `if` 语句连在一起，用来加速循环。假设 `continue` 语句用于 `while` 循环语句，要求在某个条件下跳出本次循环，其一般形式是：

```
while(表达式1)
{
    ...
    if(表达式2) continue;
    ...
}
```

这种形式的流程图如图 4.25 所示。

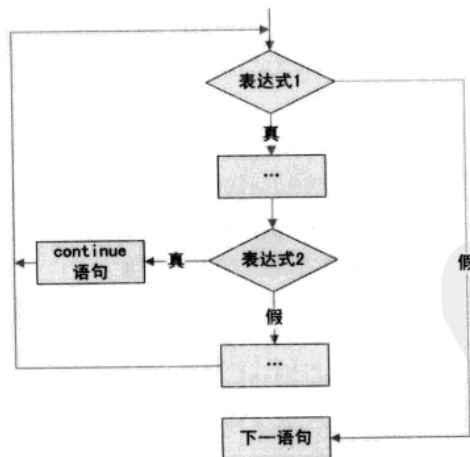


图4.25 `continue`语句用于`while`语句中的执行过程

这种形式和前面介绍的 `break` 语句用于循环的形式十分相似，区别：`continue` 只终止本次循环，而不是终止整个循环。而 `break` 语句则是终止整个循环过程，不会再去判断循环



条件是否还满足。

如果把上面判断素数的程序中的 `break` 语句换成 `continue` 语句，来看一下，程序是否还能输出预期的结果。转换后的程序清单如下所示：

```
#include <stdio.h>

void main()
{
    int i,n;
    printf("请输入一个整数\n");
    scanf("%d",&n);
    i=2;
    while(i<n)
    {
        if(n%i==0)
            continue; //跳出本次循环
        i++;
    }
    if(i==n)
        printf("%d 是素数\n",n);
    else
        printf("%d 不是素数\n",n);
}
```

运行这个程序，可能引起死循环。因为在程序执行到第 11 行的 `if` 语句时，如果满足条件 `n%i==0`，将执行第 12 行的 `continue` 语句。而这条语句的作用是：不执行后面的语句 `i++`，直接跳出本次循环。这时，`while` 语句的条件仍然被满足，程序又进入循环体执行 `if` 语句，因为 `i` 的值没有被更新，仍然满足 `if` 语句的条件，又跳出本次循环。这样，`while` 循环将永远不会终止，程序便出现了死循环。



注意：`continue` 语句不能用于循环语句以外的任何语句中。

4.5.4 goto 语句

`goto` 语句被称为无条件转向语句，其一般形式为：

```
goto 语句标号;
```

其中语句标号是一个有效的标识符，即由字母、数字和下画线组成，并且，第一个字符必须是字母或者下画线，不能是数字或者其他字符。例如：

```
goto loop_1;
```

是合法标识符，而：

```
goto 1_loop;
```

不是合法标识符。如果标识符加上一个“:”一起出现在程序中的某处，执行 `goto` 语句后，程序将跳转到该标号处并执行其后的语句。例如：

```
loop_1: printf("goto 语句跳转到此处来执行");
```

标号必须与 `goto` 语句处于一个函数中，但可以不在一个循环层中。在结构化编程语言中，`goto` 语句的“名声不好”。因为 `goto` 语句可以出现在一个函数内的任何地方，并且可以

在函数体内随意跳转，所以破坏了结构化程序单入口单出口的特点，使程序层次不清，并且不易读。来看下面的程序段：

```
...
goto loop;
int a=0;
loop: ...;
...
```

在上面的程序段中，goto 语句和其要跳转到的语句之间对变量 a 进行了定义，如果在没有使用 goto 语句的程序段中，这种方式可能被用到，而且也没有什么错误。但是，在上面的程序中，因为可能在 loop 后面的程序中要用到变量 a，而变量 a 的定义被跳过了，所以程序是错误的。很多人认为 goto 语句“非常容易被滥用”，并且建议“要谨慎使用，甚至不用”。

4.5.5 C 语言中保留 goto 语句的原因

C 语言中保留 goto 语句的原因有以下几点。

- ☑ goto 语句与 if 条件语句连用，当满足某一条件时，程序跳到标号处运行，从而构成了一种循环。

下面的程序就是一个用 goto 语句和 if 语句构成的循环，求“水仙花数”。所谓“水仙花数”是指一个 3 位数，其各个位上的数字的立方和等于该数本身。例如，153 是一个“水仙花数”，因为 $153=1^3+5^3+3^3$ 。

```
#include <stdio.h>

void main()
{
    int i,j,k,n;
    printf("水仙花数有: \n");

    n=100;
loop:  if(n<1000)
    {
        i=n/100;          //百位数
        j=(n-i*100)/10;   //十位数
        k=n%10;           //个位数
        if(i*i*i+j*j*j+k*k*k==n)
        {
            printf("%d\n",n);
        }
        n++;
        goto loop;
    }
}
```

在上面的程序中，第 9 行出现了语句 loop: if(n<1000)，其中 loop 是这行的标号。如果满足 if 语句的条件，进入 if 语句后面的复合语句，在第 19 行出现了语句 goto loop;，执行这条语句后，程序又跳转去执行第 9 行的 if 语句。这样来回跳转直到 if 语句的条件为否，从而实现循环。



用 if 语句和 goto 语句实现的循环用 while 语句、do...while 语句和 for 语句也能够实现，例如，下面的程序就用 for 语句实现求水仙花数。程序清单如下所示：

```
#include <stdio.h>

void main()
{
    int i,j,k,n;
    printf("水仙花数有: \n");

    for(n=100;n<1000;n++)
    {
        i=n/100;           //百位数
        j=(n-i*100)/10;    //十位数
        k=n%10;            //个位数
        if(i*i*i+j*j*j+k*k*k==n)
        {
            printf("%d\n",n);
        }
    }
}
```

比较两个程序，可以看出，for 语句实现的程序更好理解。所以，虽然用 if 语句和 goto 语句组合也能实现循环，但是不建议采用这种循环构造程序，而是采用 while 语句、do...while 语句或者 for 语句构造含有循环结构的程序。

- ▣ 要从多层循环的循环体中跳出到某一层循环或者整个循环体外时，使用 goto 语句可以提高效率。如果一次跳出两层以上的循环，因为 break 语句只能从内存循环退出到上一级循环，所以使用 goto 语句实现会比较方便。例如，下面的程序段用 goto 语句实现在第二层循环中出现错误时跳出到第一层循环外，执行错误代码：

```
for(...)
{
    for(...)
    {
        ...
        if(错误条件)
            goto error;
    }
}
...
error:
    处理错误的代码
```

当然，也可以不使用 goto 语句实现这段程序的功能，下面是不使用 goto 语句实现相同功能的程序段：

```
bool error_sign=false;
for(...)
{
    for(...)
    {
```



```
...
if(错误条件)
{
    error_sign=true;
    break;
}
if(error_sign)
    break;
}
...
if(error_sign)
    error:
        处理错误的代码
```

比较两段程序段可以看出，如果处理错误的代码非常重要，并且错误可能多次出现，使用 goto 语句非常方便。C 语言中保留“名声不好”的 goto 语句主要就是出于这个原因。

PDF



第5章

函 数



函数是 C 语言中支持结构化程序设计的语法基础。有了函数，在组织较大规模的程序时，问题求解的条理才更清晰、更明确。很多初学程序设计语言的人都认为函数不容易学。其实，函数的语法并不复杂，真正需要编程者去理解和体会的往往是如何用函数来拆解复杂问题。本章主要从函数的概念入手，向读者介绍 C 语言中函数的用法，以及如何利用函数来解决实际问题等内容。使用函数的水平直接反映在编程者能够组织多大规模的程序，因此这一章的知识是非常重要的，希望读者能够好好体会本章所述的内容，并力求能够灵活运用于实际开发之中。



5.1 函数与结构化程序设计

对于刚刚接触程序设计语言，特别是一开始接触的就是 C 语言的人来说，函数并不是一个非常容易理解的概念。记得笔者最开始学习程序设计语言的时候还是在小学，那个时候最先学习的计算机语言还不是 C 语言，而是 BASIC 语言（据说 BASIC 语言是一门非常适合初学者入门的语言）。

当时由于所掌握的数学知识非常有限，在头脑中还完全没有建立函数、方程等概念，因此当讲授程序设计的老师说起函数时，感觉很难理解。如果抛开数学中函数的基础，那么我们将怎样来理解程序设计语言中的“函数”？函数其实就是一个相对独立的能够完成一定具体任务的代码段。

5.1.1 函数是“黑盒子”

可以简单地把函数比喻成一个“黑盒子”，如图 5.1 所示。这个黑盒子对外只有两个接口，一个用来接收数据，另一个用来输出数据。我们只要把数据送进黑盒子，就能得到计算结果，至于盒子内部究竟是如何工作的，我们都可不必关心。函数就是这样，外部程序所知道的仅限于给函数传入什么数据，以及函数输出什么数据，其他都无关紧要。

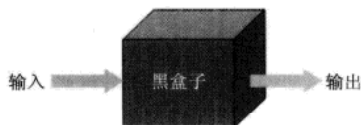


图5.1 黑盒子

不同的函数可以接收不同输入，给出不同的输出，当然这跟内部的实际工序有关，即函数所执行的功能各异。这就好比现实生活中的化学反应过程：氢气和氧气可以反应生成水，水也可以分解成氧气和氢气。

化学反应的过程总是伴随着一定物质的输入和新物质的产出，至于什么物质生成什么新物质除了跟输入有关以外，还跟反应进行的条件有关。例如，木炭在氧气中燃烧可以生成二氧化碳，但是在某些条件下也可能产生一氧化碳，如图 5.2 所示，这取决于实际反映的条件。函数也是这样，同样的输入，也可以得到不同的结果，这跟函数内部的实现有关。

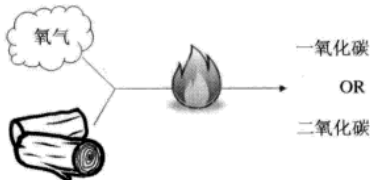


图5.2 化学反应与其反应条件有关

经过上面的描述读者应该对函数有了一个初步的认识，可以明确函数就是接收输入，并在其内部处理数据，最后输出结果是一个独立的代码单元。



5.1.2 数学函数与 C 语言函数

下面我们试图从数学函数的概念来进一步解释程序设计中函数的意义。数学函数与 C 语言函数的确有很多相似之处。请读者来看下面这样一个简单的数学问题： $y = 2x + 3$ ，其中 $x = 3k^2 + 1$ ， $k = 0$ ，求 y 的值。这个问题非常简单，只要将变量 k 的值带入第二个函数求得 x 的值再进一步带入第一个函数就可求出 y 的值。这个过程跟 C 语言中的函数调用过程极其相似。

1. C 语言中的 main() 函数

前面谈到过主函数这个概念，主函数即 `main()` 函数。C 程序总是从 `main()` 函数开始执行的，`main` 函数可以调用任何 C 语言函数，但任何 C 语言函数都不能调用 `main()` 函数，`main()` 函数通常在其所属的程序执行时被操作系统调用。`Main()` 函数可以没有参数也可以有参数，如果不需要从命令行中获取参数，则使用无参的 `main()` 函数，否则使用有参的 `main` 函数。方程 $y = 2x + 3$ 相当于是程序的主函数，而 $x = 3k^2 + 1$ 则相当于是被 `main` 函数调用的一个普通函数。

2. C 语言函数的组成

函数总是由 4 个部分构成的，即函数名、参数、返回值和函数体。

☑ 函数名：

函数名是必需的，它声明了函数的存在，没有函数名就不存在函数。函数的参数、返回值和函数体都可以为空。

☑ 参数：

参数又分为形式参数和实际参数（关于这两种参数的内容本书后面还有详细介绍）。形式参数相当于数学函数的自变量，实际参数相当于为自变量所取的确定值。仍以前面的数学问题为例，在 $x = 3k^2 + 1$ 函数中，自变量 k 就相当于是函数的形式参数，而 $k = 0$ 中的 0 就是函数的实际参数。

☑ 返回值：

当形式参数获得确定的值时，经过一定的计算后得到的要返回给调用函数的确定值就是返回值，例如函数 $x = 3k^2 + 1$ 返回给函数 $y = 2x + 3$ 的计算结果，即 x 值，就可以看做是该函数的返回值。上述过程如图 5.3 所示。

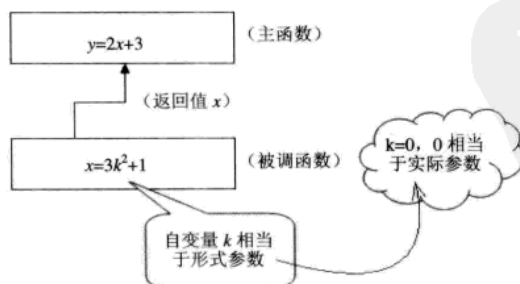


图5.3 函数的调用解析


☑ 函数体:

函数体用来完成函数的具体功能,可以为空。

5.1.3 C 语言函数中的库函数

事实上,读者在本书前面已经接触过很多函数,例如 `printf()`、`scanf()` 等。我们称它们是库函数,也就是系统中已经编制好的,可以直接调用的函数。

不同的编译器会提供不同的库函数,每个函数都可以完成一定的功能,从而帮助用户搭建复杂的程序。尽管各个编译器所提供的函数不同,但所有编译器基本上都可提供输入\输出函数、数学函数、字符串函数、内存函数、文件函数等在内的几大类常用函数。

 **提示:** 在使用 C 语言进行编程时,如果能熟练使用这些库函数,那么编程开发将事半功倍。当然,在使用这些库函数之前,必须显式地引用其所在的头文件。这样,程序连接时,编译器就会自动从相应的库中装配所需程序。

5.1.4 结构化的程序设计

除了使用系统已经提供的库函数以外,也可以自己编制函数。系统所提供的库函数是用来完成基础性的通用功能的,而我们自己编写的函数则是用来完成针对特定开发目的的专用功能的。

在这一章中,我们更多关注的是用户自己编写函数的方法及原则。这些函数能够极大地提高程序质量,简化开发的复杂度,这也是使用函数的好处。当然,也可以笼统地说使用函数的好处就是能够实现结构化的程序设计,所以其他一些具体好处都是由结构化程序设计衍生发展而来的。

1. 结构化程序设计

结构化程序设计的思想是指以模块化设计为中心,将待开发的软件系统划分为若干个相互独立的模块。每个模块彼此独立,相互不会影响。因此这样更便于把一个复杂的问题分解为若干个独立的小问题,这样问题就将更加明确而易于实现,从而为整个开发过程奠定一个健康的流程。

2. 函数在结构化程序设计中的角色

函数是实现结构化程序设计的基础,每一个函数都可以看作是一个独立的模块,它们将独立完整的功能单元进行有效封装,从而对复杂系统的搭建和设计进行了简化。这也更便于大规模的流水作业。

想象一下现如今的工厂是如何工作的,无论是一个生产飞机的工厂,还是一个生产电子琴的工厂,它们都拥有一条长长的流水线。流水线上的工作仅仅是进行零件的组装。而一件完整的产品往往被分割成许多独立的部分。

例如,飞机的发动机、机翼、座椅等都是单独生产的,生产发动机的部门只要为最后组装提供发动机即可,而无须关心机翼的形状如何。最后将所有的零件有序地组装到一起,一架看似复杂无比的飞机就这样被轻松地制造出来。



3. 结构化程序设计的好处

结构化程序设计除了明确问题本身，并简化问题的求解以外，还有以下好处：

☐ 结构化程序设计方法提高了软件的复用率。

就像库函数一样，一个 `printf()` 函数被编写好之后，很多程序再需要进行输出操作时就不用再重复编写该功能函数。我们为某个软件所编写的函数完全可以移植到另外一个软件中，只要需求相同，这些函数就可以共享。

☐ 结构化程序设计提高了程序的可读性，便于排错和维护。

如果一个长达上万行代码的程序都被写在一个 `main()` 函数中，那时我们的程序一旦测试出错，那么找到错误的过程无异于大海捞针。而如果程序被分成许多精细的小函数，那么定位就要迅速得多。

这就好比一个管理井井有条的仓库，不同的东西会被分类摆放到不同的房间中，我们需要什么东西只需到指定的存储房间去寻找即可。但如果将所有物品都堆在一起，那么找到一样东西就会变得很复杂。

☐ 结构化的程序设计方法对于软件后期的升级维护也大有裨益。

例如，一台计算机随着时间的推移可能硬盘显得有些不够大，这时我们只需换上一个新硬盘即可，而不用把整台计算机都换掉。同理，当一个软件系统需要升级的时候，可能我们需要做的只是升级它的某一个部分，而不一定要把整个软件都重新编写。

这时如果程序被非常有条理地划分为几个独立的模块，那么我们就只需更换需要升级的模块即可。杀毒软件是现实中一种需要频繁更新的软件系统，这样它才能够应对层出不穷的病毒攻击。绝大多数更新只是在对杀毒软件的病毒库进行更新，而不是把整个软件都重新安装，这就是一个非常典型的说明结构化程序设计好处的例子。

5.2 函数的使用

从本节开始，我们将正式进入函数的学习。要正确地使用函数，首先要掌握函数声明和调用的语法，这也是本节所要研究的重点。关于函数结果返回以及参数传递方面的问题是决定读者能否深刻理解函数用法的核心部分。尤其在参数传递方面，还涉及变量作用域的问题，这些地方都是需要读者小心留意的。

5.2.1 函数的定义

函数定义的一般形式如下：


```
类型标识符 函数名(形参列表)
{
    函数体;
}
```

上述定义中，形参列表、函数体和类型标识符都是可选项。其中类型标识符表示的是函数返回值的类型，也就是意味着函数将带回一个哪种类型的数值。函数也可以没有返回值，此时，通常用关键字“`void`”来标识，如下例中的函数 `greeting` 没有返回值，在它的函数名前，我们就用 `void` 来修饰。

```
#include <stdio.h>

void greeting()
{
    printf("Good morning!\n");
}

void main()
{
    greeting();
}
```

 注意：在 Visual C++ 6.0 中，如果函数没有返回值，关键字 void 也可以省略，但这时编译器会抛出一个警告，告诉读者或许该函数需要一个返回值。尽管进行了警告但并不影响程序的编译和运行，我们建议读者在不需要函数返回任何值时，使用关键字 void 来修饰函数名，这是一种比较好的编程习惯。

5.2.2 函数的类型

依据函数是否需要传递参数，函数可以分为：

☐ 有参函数。

☐ 无参函数。

当函数没有参数的时候，其括号中的形参列表可以什么都不写，也可以写一个关键字 void，但是括号不能够被省略，括号是用来区别函数声明与变量声明的重要标志。

对于有参函数，其形参列表采用的是如下一种语法方式：

类型标识符 函数名(类型标识符 1 变量名 1, 类型标识符 2 变量名 2...);

其中每个形式参数前面都需要指定一个数据类型。例如，下面给出的这段示例代码中定义了一个用来计算两个整数之和的函数 add()，它使用两个形式参数 x 和 y 来作为参与加法运算的两个加数，参数 x 和 y 都是整型的。该函数将计算结果作为返回值，return 语句的作用就是将计算结果返回。此外，需要说明的是 return 语句中的括号也是可以省略的。

```
#include <stdio.h>

int add(int x, int y)
{
    int result;
    result = x + y;
    return (x+y);
}

void main()
{
    int sum = add(30, 50);
    printf("%d\n", sum);
}
```

前面已经说过，函数定义中的函数体是可选的。例如：

```
#include <stdio.h>
```




```
void display() {}

void main()
{
    display();
}
```

上述示例代码中函数 `display()` 的函数体为空，表示该函数什么都没做。C 语言中允许这样的空函数存在。

通常来说，空函数是没有意义的，因为它并不进行任何操作。但在程序设计过程的初期，由于程序设计尚处于雏形阶段，某些函数尚未完全实现，因此这时可以用一些空函数来占据一定位置，随着软件程序开发工作的展开与深入，以后再将这些空函数填写完整。在设计初期使用空函数，能够使程序的结构更加清楚，可读性更好，扩充更方便，因此我们也建议读者在组织比较复杂的程序时恰当、灵活地运用空函数。

5.2.3 函数的返回值

当函数被调用时，函数体内往往会进行一些操作或计算，如果希望将这些操作或计算结果返回给主调函数，那么就需要使用函数的返回值。函数的返回值是被调函数与主调函数之间进行信息沟通的重要途径，如图 5.4 所示。

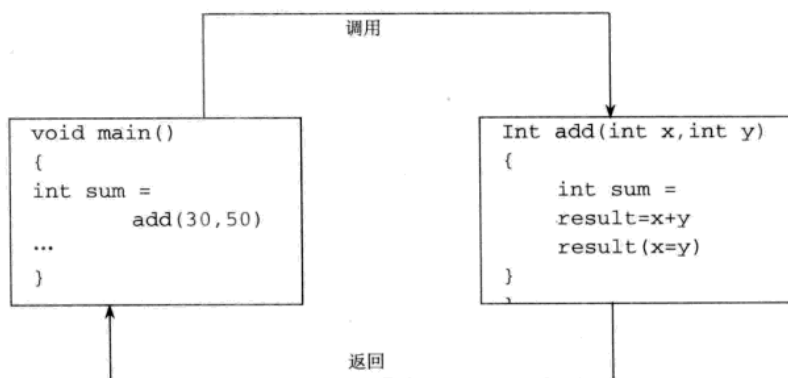


图5.4 函数调用与返回

1. return 语句的使用原则

- ☑ 函数的返回值是通过 `return` 语句返回的，该语句可以将一个确定的值返回到主调函数中，并且该值可以是任意类型的。
- ☑ 如果被调函数是无返回值的，那么函数体内可以有 `return` 语句，也可以没有 `return` 语句。但是如果被调函数是有返回值的，那么函数体内就必须包含 `return` 语句。
- ☑ 一个函数中可以包含有多个 `return` 语句，执行到第一条 `return` 语句时，函数都将返回到主调函数中，此后的其他语句则不会继续执行。

例如, 如果 a 大于 b , 那么函数将在第一个 `return` 语句处返回, 剩余的代码将不会被执行。同理, 如果 a 小于 b , 那么函数将在第二个 `return` 语句处返回, 剩余的代码则不会被执行:


```
int judge(int a, int b)
{
    if(a-b > 0)
        return 0;
    else if(a-b < 0)
        return 1;
    else if(a-b == 0)
        return 2;
}
```

☑ 对于无返回值的函数, 由于其中可能不包含 `return` 语句, 因此在这种情况下 (即函数体内没有 `return` 语句的情况), 函数体内的语句将逐条执行, 直到表示函数结尾的大括号处。

☑ `return` 语句后面可以是一个常量, 也可以是变量, 还可以是一个表达式, 但无论是常量、变量还是表达式, 它们都必须具有一个确定的值。

例如, 下面这些 `return` 语句都是正确的:

```
#define PI 3.14
int x = 0;
int y = 1;
return (0);
return 0;
return (PI);
return (x-y);
return (x>y ? x : y);
```

 注意: 前面已经提过 `return` 语句后面的括号是可选的, 因此去掉上述语句中的括号也是正确的。

☑ 函数声明语句中函数名前面的“类型标识符”表示该函数返回值的类型。如果函数有返回值, 那么这个返回值必将属于某种确定的类型。但是 C 语言中也规定: 凡是不加类型说明的函数, 一律默认为返回值类型为整型。


例如下面这段程序也是正确的:

```
#include <stdio.h>

display(int x, int y)
{
    return x>y ? x : y;
}

void main()
{
    printf("%d\n", display(3, 4));
}
```



 注意：尽管这样做并不会引起语法上的错误，但我们仍然建议读者在声明函数时显式地指定函数的返回值类型，这样能够使程序的可读性更强，更利于调试和维护。

- ☐ 在定义函数时，对于函数类型的说明应该与 `return` 语句中表达式所给出的值类型一致。也就是说，如果函数声明的类型是整型的，那么函数中 `return` 语句所返回的值也应当是整型的。

2. 返回值类型与函数类型不同的 3 种情况

如果返回值类型与函数类型不同，情况又会怎样呢？总的来说有 3 种情况。

- ☐ 第一种情况就是“并无影响”。

看下面这段示例代码：

```
#include <stdio.h>

int func1()
{
    int x;
    scanf("%d", &x);
    return x;
}

void main()
{
    char c;
    c = func1();
    printf("%c\n", c);
}
```

由于字符在计算机内部都是以 ASCII 码形式存储的，因此字符在计算机看来就是一些整数，所以上述代码中尽管返回值类型与函数的声明类型不同，但程序的运行并无影响。

- ☐ 第二种情况，称为“精度损失”。

看下面这段示例代码：

```
#include <stdio.h>

int func2(float x, float y)
{
    float bigger;
    bigger = x > y ? x : y;
    return bigger;
}

void main()
{
    int result;
    result = func2(3.14159, 2.71828);
    printf("%d\n", result);
}
```

编译并运行上述程序，程序输出的结果是整数“3”。这是因为函数 `func2` 被定义为整型，

而 `return` 语句返回的是一个实型，于是函数在返回时会将实型数据转换为整型，即 3.14159 就变成了 3，精度被损失了。读者应该明确，精度的损失是在函数返回时就已经发生的，因此即使改写 `main()` 函数如下，程序的输出结果也不会是 3.14159，而应该是 3.00000：

```
void main()
{
    float result;
    result = func2(3.14159, 2.71828);
    printf("%f\n", result);
}
```

▣ 第三情况，称之为“编译出错”。

也就是说程序完全无法通过编译，其实这种情况更为多见，这里我们就不再举例了。但是通过上述几点的讨论，读者应该不难看出，函数类型的说明与 `return` 语句中表达式所给出的值类型不一致时，可能引发的情况是非常复杂多变的。这不仅会导致程序得出一些古怪的结果，还会给程序的调试和维护带来麻烦，因此我们强烈建议读者在使用函数时要显式地声明函数类型，并保证函数类型的说明与 `return` 语句中表达式所给出的值类型相同。

5.2.4 函数的参数

在函数调用过程中，主调函数和被调函数之间往往需要进行一定的信息沟通，除了上文中所讲的返回值以外，参数传递也是用来实现这种信息沟通的重要途径。

1. 参数分类

参数分为形式参数和实际参数。形式参数是指定义函数时，函数名后面括号里的变量名；而实际参数则是指主调函数调用被调函数的语句中，函数名后面括号里的参数表达式。来看下面一段示例代码：

```
#include <stdio.h>


int area(int length,int width)
{
    return length*width;
}

void main(){

    int result=0;
    result=area(30,40);

    printf("The area of the rectangle is %d.\n", result);
}
```

上述程序中声明了一个函数 `area()`，它用来计算一个矩形的面积。该函数声明中指定了两个形式参数 `length` 和 `width`，它们位于函数声明语句中函数名后面的括号里。

 **注意：**形式参数声明时前面的数据类型不能省略。

再来看 `main` 函数中的 `area` 函数的调用语句，括号里面的 30 和 40 就是实际参数。函数



area()在被 main()函数调用时,形式参数 length 被赋予整数值 30,而 width 则被赋予整数值 40。通过这样一种参数传递方式,主调函数和被调函数之间进行了有效的信息传递。



注意: 实际参数可以是值,也可以是用来表示值的表达式。

2. C 语言中的参数传递机制

C 语言中的参数传递机制采用的是一种被称为“按值传递”的方式。当主调函数调用被调函数时,系统会把实参的值赋给形参,而且这个过程是绝对单向的,也就是说值能且仅能从实参传给形参,反过来则不行。

形参与实参的实际存储单元不同,形参仅相当于实参的一个副本。在调用函数时,被调函数内部的形参会被临时分配一个存储单元,当函数返回后,这个临时单元就会被释放,于是形参的生命周期也就结束,在程序中继续使用形参是无效的。

而且由于形参仅仅是实参的一个副本,即二者并不共享同一存储单元,因此,形参的任何改变对于实参都毫无影响。当函数返回后,形参被销毁,而实参则维持不变。例如下面这段示例代码:

```
#include <stdio.h>

void swap(int a, int b, int temp)
{
    temp = a;
    a = b;
    b = temp;
}

void main()
{
    int a, b;
    a = 30, b = 40;
    swap(a, b, 0);
    //temp = 100; !!!ERROR
    printf("%d\n", a);
}
```

3. 参数传递过程

图 5.5 说明了上述代码里参数传递过程中实际发生的事情。

第 1 步 当 main()函数通过语句“swap(a, b, 0);”调用函数 swap()时,main()函数中变量 a 和 b 的值被传进了函数。这时计算机会为形式参数 a 和 b 重新开辟一块存储单元,相对应地,形式参数 a 获得了一个值 30,形式参数 b 获得了一个值 40。



注意: 形式参数 a 和 b 与 main()函数中变量 a 和 b 并不一样,它们被分别存放在不同的地方,图 5.5 中以不同的颜色来标识。

第 2 步 函数 swap()中将 a 和 b 的值进行了对调。于是在数字标识 3 的地方,形式参数 a 的值变为了 40,形式参数 b 的值变为了 30。在此过程中 main()函数中的变量 a 和 b 的值却并未改变。

第3步 当函数调用结束返回后，形式参数 *a* 和 *b* 的存储单元就会被释放，在数字标识 4 的地方被销毁。而 *main* 函数中变量 *a* 和 *b* 仍然保存着原来的值，因此当 *main* 函数中使用 *printf* 语句将变量 *a* 的值输出时，得到的是 30，而非 40。

注意：一旦返回主调函数，被调函数中的形式变量就已经不再存在了，因此如果将上述代码中被注释掉的语句“*temp* = 100;”还原，编译器就会报错。这是因为变量 *temp* 只是被调函数中使用的一个形式参数，它在函数返回后已经不存在了。

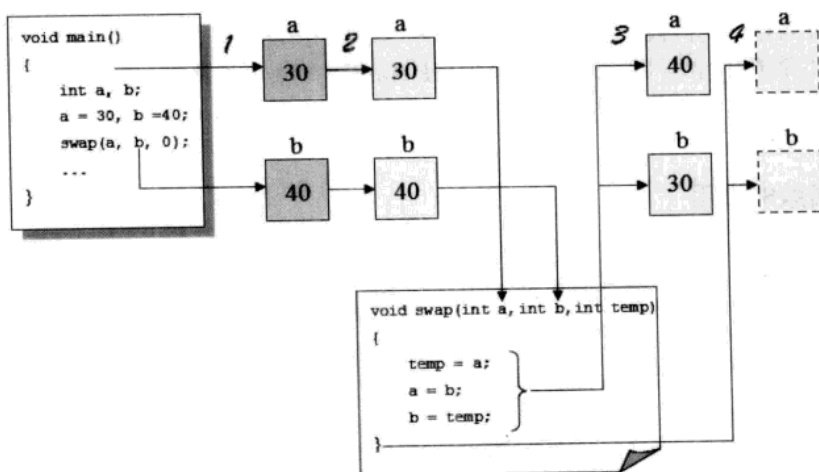


图5.5 参数传递

实参除了可以是常量以外，还可以是变量或者表达式，但要保证这些表达式或者变量都必须拥有确定的值。例如，以下示例代码中 *main()* 函数里的几个函数调用形式都是正确的：

```
#include <stdio.h>
#define NUM 10

int func(int x, int y)
{
    int bigger;
    bigger = x > y ? x : y;
    return bigger;
}

void main()
{
    int a, b, c;
    a = func(30, 40);
    b = func(NUM, 3);
    c = func(a-b, NUM*5);
    printf("%d, %d, %d\n", a, b, c);
}
```




4. 实参与形参的列表匹配问题

在函数调用语句中，实参列表中变量及其类型应该与函数声明语句中形参的列表相对应，也就是说对应的实参与形参的类型应相同或赋值兼容。例如前面那段程序中，函数调用语句“func(30, 40);”使用了两个整数，而函数 func() 声明参数列表中所给出的对于形参类型也是整型，因此这样做就是正确的。

除了字符型与整数可以通用以外，其他情况的类型不匹配要么是引起精度损失，要么就会引起编译错误。即使出于某种特殊的要求而刻意希望参数列表类型并不精确对应，也必然会引起程序可读性的降低，不利于调试和维护。其他的情况则会导致编译出错，这些情况都是我们所不希望看到的。因此，建议读者在使用函数的时候，严格遵守参数列表中对应的实参与形参的类型应当相同的原则。无论何时，正确、规范地使用函数都是最大程度地发挥其作用与优势的前提。

5.2.5 函数的调用

前面的例子中已经给出很多函数调用的实例，但是要用好函数可能还并不容易。这里我们将对函数的调用进行一个系统性的介绍。

1. 函数调用的形式

根据前面的一些经验，可以总结出函数调用的一般形式，其中，实参列表可以为空，但括号不可省略：

函数名(实参列表);

2. 函数调用的方式

在实际使用过程中，函数调用的方式又可分为以下 3 种情况。

- ☑ 函数调用可以作为一个语句单独使用，这时函数有没有返回值都可以。而更多的时候，单独把调用函数作为一个语句往往只是借由函数来完成一定的操作而已。例如 5.2.1 小节中第一个程序示例的函数 greeting() 就属于这种情况。
- ☑ 函数可以出现在表达式中，即作为表达式的一部分来使用，这时函数必须返回一个确定值以参与运算。例如下面的程序段就是属于这种类型的：

```
//函数声明
int f(int x, int y);
int a = 100;
int b = 99;

int sum = 5 + f(3, 1);
int mean = (sum + f(a+b, a-b))/2;
```

- ☑ 函数调用可以作为实参来使用。例如 5.2.3 小节的示例代码里给出的语句：
“printf(“%d\n”, display(3, 4));”，这条语句就是把函数 display 的调用当做是 printf() 函数的实参来使用的。

3. 编译环境决定实参的求值顺序

函数调用时，如果函数是带有多个参数的，那么实参列表中各个参数之间需用逗号隔开。而且实参列表要与形参列表的参数数目及类型上一一对应。但是对于实参列表中包括多个参

数的情况，各实参的求值顺序有时却不尽相同。这完全跟所使用的编译环境有关，有的环境系统按自左向右的顺序对实参进行求值，而有的环境系统则按自右向左的顺序对实参进行求值。这对程序的计算结果影响很大，例如：

```
#include "stdio.h"

int func(int a, int b)
{
    return a;
}

void main()
{
    int i = 1;
    printf("%d\n", func(i, ++i));
}
```

在 Visual C++ 6.0 下运行上述程序，该程序输入的结果是“2”，这表明 Visual C++ 6.0 开发环境所遵循的求值顺序是自右向左的，于是函数调用语句“func(i, ++i);”就相当于“func(2, 2);”，所以最后返回值是 2。

但是如果遵循从左向右的求值顺序的话，函数调用语句则相当于“func(1, 2);”，那样的话返回值就应该是 1。这很令人困惑，因为不同的环境对于同一语句的解释规则有可能不一样，所以这里要提醒读者尽量避免使用这种语句，从而规避它的二义性所带来的混淆。

4. 函数的调用出错

关于函数的调用还有最后一个问题要向读者说明，为此请先来看一段示例代码。这段程序与上述程序基本相同，但是我们将 main() 函数与函数 func() 的顺序调换了一下，请读者在 Visual C++ 6.0 下编译这段程序：

```
#include "stdio.h"

void main()
{
    int i = 1;
    printf("%d\n", func(i, ++i));
}

int func(int a, int b)
{
    return a;
}
```

结果编译器抛出了编译错误：“error C2065: 'func': undeclared identifier”，这是一个很值得我们去探讨的现象。编译器抛出的错误提示指出系统无法识别 func 标识符。这是因为编译器在解读程序时，总是按照从前往后、从上往下的顺序逐条解读语句的。

当解读到语句“printf("%d\n", func(i, ++i));”时，由于标识符 func 是首次出现，编译器当然就无法识别它了，对于编译器来说标识符 func 是一个完全陌生的符号，所以编译器就抛出了错误。



5. 函数调用出错的解决办法

当函数调用出错时有以下两种解决办法:

- ☑ 按照程序最初的方式,即将函数 func()的定义置于 main()函数之前,程序即可以顺利地编译通过。
- ☑ 在使用这个函数前就向编译器“介绍”它,这种向编译器介绍新函数的方法称为“声明”。

6. 声明和定义

声明和定义是不同的,但它们又很容易被混淆。声明的作用是把函数的名字、类型及参数情况告知编译系统,而这个过程并不会生成函数的实体。打个比方来说,声明就相当于一个人将他的户口调入了一个新的居住地,对于派出所和居委会来说已经知道有这么一个人会住进小区里。但是这个人实际搬没搬进来住却不一定。

“定义”一个函数则意味着把函数完全实现,包括给出完整的函数体内容。一旦函数被定义,计算机就会为其开辟存储空间,把函数实际写入存储区。这就相当于某个人不仅把户口调入了新居住区,而且这个人也确实搬入了小区居住。

函数声明的一般语法形式如下:

类型标识符 函数名(类型标识符 1 变量名 1, 类型标识符 2 变量名 2...);

或者:

类型标识符 函数名(类型标识符 1,类型标识符 2...);

也就是说,在声明函数的时候,形参列表中只需给出参数的类型即可,至于参数名则可有可无。例如,可以改写前面的代码片段如下:

```
#include "stdio.h"

int func(int, int);

void main()
{
    int i = 1;
    printf("%d\n", func(i, ++i));
}

int func(int a, int b)
{
    return a;
}
```

以上示例中将函数 func()声明为一个全局的函数,因此只要是位于该文件中的函数都可以调用此函数。另外,也可以将函数的声明放在主调函数中,这样,在函数定义出现之前,将只有包含其声明的函数才能调用它。修改后的代码片段如下:

```
#include "stdio.h"

void main()
{
```

```

int func(int, int);
int i = 1;
printf("%d\n", func(i, ++i));
}

int func(int a, int b)
{
    return a;
}

```

除了对于函数声明与定义之间区别的理解之外,此部分内容还涉及函数作用域的问题,关于作用域的内容将在本章后面进行更加详细的介绍。

5.2.6 函数的嵌套——蒙特卡罗法求圆周率 π

主函数中可以调用其他函数,同样,其他函数也可以再调用另外的函数。在 C 语言中只有 main 函数不能被普通函数所调用。这种在函数体内调用其他函数的行为就称为函数的嵌套调用。

为了配合函数嵌套调用的讲解,这里举一个非常有名,也非常有意思的蒙特卡罗法求圆周率 π 的例子。从数学的角度来看,赌博就是一种随机实验。后来人们就把通过随机实验方法进行问题求解的一类概率算法称为蒙特卡罗法。

如何使用蒙特卡罗法来求 π 值? 现在假设你是一个赌徒,离你不远处的一面墙上有一个正方形的靶子,靶子中间有一个圆形的靶面,如图 5.6 所示,圆形刚好与正方形边框的四条边相切。

假设你没有专门练习过飞镖投掷,现在让你随机向圆形靶面上投掷飞镖,如图 5.7 所示。

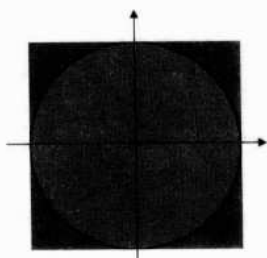


图5.6 圆形靶面



图5.7 投掷飞镖



我们知道圆的面积公式为 $S=\pi R^2$, 当 $R=1$ 时, $S=\pi$ 。如果在一个边长为 2 的正方形中有一个内切圆,那么该圆的面积就为 π 。利用这一原理,可以假设有大量随机点等概率地、均匀地落入正方形中。也就是当你随机向靶子上投掷大量飞镖时,假设所有飞镖都落入正方形的靶面上。那么可知:

$$\text{内切圆面积 } S / \text{正方形面积 } S' = \text{落入圆中的点数 } n / \text{随机点的总数 } N$$

由此即可得出 $\pi = 4 \times n / N$ 。基于上面的描述就可以设计一个计算机程序来模拟飞镖的投掷过程。由计算机随机生成一些点,这些点的坐标为 (x, y) , 其 $-1 < x < 1$, $-1 < y < 1$ 。如果内切圆的半径为 r ,则根据圆形的几何定义可知:当 $z = \sqrt{x^2 + y^2} > r$ 时,点落在了圆形靶面的外部,



即未命中；当 $z = \sqrt{x^2 + y^2} \leq r$ 时，点落在了圆形靶面的内部，即已命中。

根据上述分析，完成代码如下：

```
#include "stdio.h"
#include "stdlib.h"
#include <math.h>
#include <time.h>

//模拟飞镖的投掷过程
bool toss()
{
    double x = 0.0, y = 0.0;
    double sign = ((double)rand()/RAND_MAX);
    double size = ((double)rand()/RAND_MAX);
    if (sign > 0.5)
        x = size;
    else
        x = -size;

    sign = ((double)rand()/RAND_MAX);
    size = ((double)rand()/RAND_MAX);
    if (sign > 0.5)
        y = size;
    else
        y = -size;

    double z = sqrt(x*x + y*y);    //确定是否命中

    if(z>1)
        return false;
    else return true;
}

//计算投掷命中率
double ratio(int num)
{
    int i, sum = 0;
    srand((unsigned)time(0));    //设置随机种子
    for(i = 0; i < num; i++)
    {
        if (toss() == 1)
            sum++;
    }

    return (double) sum / (double) num;
}

//蒙特卡罗法求圆周率
```

```
double MonteCarloPi(int num)
{
    return ratio(num)*4;
}

void main()
{
    double pi = MonteCarloPi(10000);
    printf("PI = %f\n", pi);
}
```

读者可以完成编码后编译并运行程序，设置一个随机实验次数，来看看程序计算出的圆周率近似值是否正确。

注意：概率随机实验求得的仅仅是近似值而非准确值，而且由于每次输入的随机种子不同，因此获得的结果也不会相同。

上面这个程序是一个典型的函数嵌套调用的实例。如图 5.8 所示，程序从 main()函数开始执行。

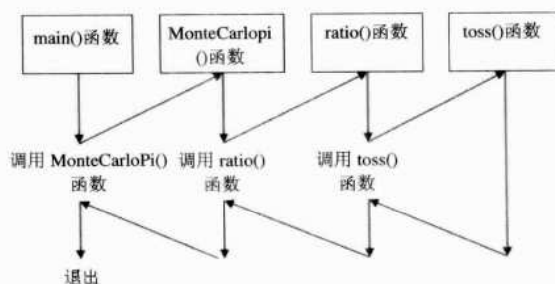


图5.8 函数的嵌套调用

main()函数中调用了 MonteCarloPi()函数，而 MonteCarloPi()函数又调用了 ratio()函数。Toss()函数模拟的是投掷飞镖的过程，因此如果设定投掷次数为 N ，则 ratio()函数就会调用 toss()函数 N 次，注意这是通过 ratio()函数体中的 for 循环来实现的。当被调函数执行结束后，就会返回到主调函数中相应的位置，继续执行主调函数中的下一条语句。嵌套调用的函数层层返回，直到 main()函数中。最后当 main()函数执行完毕后，程序退出。

提示：尽管 C 语言中允许函数之间进行嵌套调用，但是并不允许函数的嵌套定义。也就是说一个函数内不能包含另外一个函数的定义，这一点请读者务必注意。

5.3 递归

递归是强大的问题求解工具，是程序设计中的一种重要思想和机制。函数直接或间接地调用其自身就形成了递归调用。递归有助于写出清晰易懂的代码，能有效提高程序的整体风格。下面将从递归的概念入手，向读者介绍使用递归的一些基本原则，同时给出了一些应用递归进行求解的经典实例。



5.3.1 递归的定义

理解递归的定义非常重要。通常，在数学及程序设计方法学中为递归下的定义是这样的：若一个对象部分包含它自己，或用它自己来定义自己，则称这个对象是递归的；若一个过程直接或间接地调用自己，则称这个过程为递归的过程。可见，对于函数调用来说，如果某个函数直接或间接地调用其自身，那么这个过程就是一个递归的调用过程。恰当的应用递归方法可以使一些复杂的问题得以简化。

1. 递归在日常生活中的例子

在日常生活中，递归的例子是十分普遍的。下面简单举几个例子来阐释递归的概念。第一个很容易想到的例子就是平行镜子成像的例子，试着想想看，当两面镜子 A 和 B 以几乎平行的角度相对摆放时，一面镜子 A 将映出另外一面镜子 B 中的景象，而另一个镜子 B 中的景象正是镜子 A 自身，如此往复，相互嵌套的图像就形成了一种形式的递归。

字典是生活中另外一个常见的递归实例。字典中任何一个词汇都是用“其他的词汇”解释或定义的，但是“其他的词汇”在被定义或解释时又会间接或直接地用到那些由它们定义的词。使用者可以读懂字典中全部词汇的释义，前提是使用者必须掌握一些少量的非常基础的词汇的含义。当使用者查询某个生僻的词汇时，他发现释义中的某个词汇无法理解，因此他继续在字典中查询那些暂时不能理解的词汇的释义。如此继续下去，但此过程最后必然终止，因为最后所有的释义都归结为读者已经了解的常见意思，否则字典使用者将陷入一个无解的问题陷阱中。

2. 递归在数学上的应用

数学上常用的阶乘函数、斐波那契序列等，它们的定义和计算都是递归的。以阶乘函数为例，其定义如下：

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

这个定义十分容易理解而且易于使用，由此可见，每个正整数的阶乘都等于它自身乘以比它小 1 的那个数的阶乘，而 0 的阶乘则为 1。下面来看看 4 的阶乘是如何根据上述定义进行求解的：

```
4! = 4 × 3! = 4 × (3 × 2!) = 4 × 3 × (2 × 1!)
    = 4 × 3 × (2 × (1 × 0!)) = 4 × 3 × (2 × (1 × 1))
    = 24
```

下面给出了阶乘的递归求解函数：

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else {
        int value = factorial(n - 1);
        return n * value;
    }
}
```


递归在描述和解决实际问题时，能够令问题求解更清晰。除此之外，递归的过程也具有简洁、易编、易懂等诸多优点，因此在实际开发中颇受青睐。

5.3.2 使用递归的原则

定义递归方法时务必谨慎，因为不适当的递归很可能让一个函数永无止境地调用其自身，这是我们所不期望见到的。正确使用递归需要遵循一系列基本原则。

1. 确保递归向着基本条件进行

编写递归函数时，首先必须保证有一些“基本条件”能够采用非递归的方式计算得到，这是使用递归方法的重要前提。所谓“基本条件”就是指当采用递归处理后的子问题可以直接解决时，就停止分解，这些可以直接求解的问题叫做递归的“基本条件”。为了使计算最终能够完结，任何递归调用都要向着“基本条件”的方向进行。

回头看看前面的阶乘求解函数，其中语句“if (n==0) return 1;”就是所谓的基本条件，因为当整数值为0时，其阶乘就不再采用递归的方法来求解了，如果阶乘函数中不存在这样一个引导递归走向结束的条件，那么这个递归函数就将永无止境地进行下去。

下面来看一个用递归实现的二分查找函数，试着找出其中的“基本条件”。在给出代码清单之前，先来看看什么是“二分查找”。二分查找又称为折半查找，它的基本思想对于一串有序元素，先确定待查元素的范围，并将有序元素序列分成两半，然后比较位于中间点元素的值。如果该待查元素的值大于中间点元素的值，则将范围重新设定为大于中间点元素的范围，反之亦然。与传统的顺序查找算法相比，折半查找是一种更加快速的搜索方法。这种方法对集合中元素的个数没有限制，即使是很大的集合，折半搜索也能迅速地找出特定元素。虽然折半查找对集合中元素的个数没有限制，但它要求元素序列预先是有顺序排列的。下面将结合图示介绍折半搜索的工作原理。

假设现有集合A如图5.9所示，其中的元素已按从小到大的顺序排列。

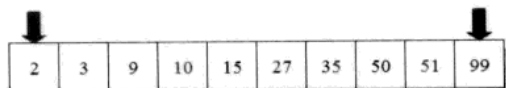


图5.9 集合A

现要应用折半查找法搜索值为51的元素。首先算法用数值51与集合中位于正中间的元素值进行比较。由于在本例中，集合元素是偶数个，于是位于中间点的元素就有两个，这里可以规定出现这种情况时取其中较小的那个。可见，算法检测到位于集合正中间的元素值为15。由于此处元素按照从小到大的顺序排放，所以值为51的元素应当位于值为15的元素的右边。算法放弃集合的左半部，转而在集合的右半部中进行搜索，所以左端的指针现在指向了元素15右端的一个值，即27。第一次比较后，搜索范围被缩小了，如图5.10所示。



图5.10 第一次比较



在右半集合中的搜索方式与上述搜索方式相同。首先，算法用值 51 与右边集合中位于正中间的数据（此处为 50）进行比较。如图 5.11 所示，这时算法检测到此位置中元素的关键码为 50。由于 51 大于 50，所以这时算法忽略子集合的左半部，转而在子集合的右半部中进行搜索。搜索过程与上述过程类似。经过 3 次比较，算法最终找到值为 51 的元素在集合中的位置。

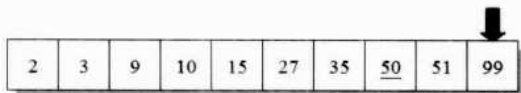


图5.11 第二次比较

上述过程中，虽然集合中有 10 个元素，但算法只用了 3 次比较就找到了目标元素，可见折半查找算法的效率很高。下面就给出基于递归实现的折半查找函数的示例代码，代码中含有数组的内容，建议理解尚有困难的读者可在完成下一章数组部分的学习后再回来研究这段程序：

```
#include "stdio.h"
#define SIZE 10

int binarySearch(int num[SIZE], int value, int low, int high)
{
    if(low > high)
        return -1;
    int mid = (low + high)/2;

    if(value < num[mid])
        return binarySearch(num, value, low, mid-1);
    else if(value > num[mid])
        return binarySearch(num, value, mid+1, high);
    else
        return mid;
}


void main()
{
    int array[SIZE] = {2, 3, 9, 10, 15, 27, 35, 50, 51, 99};
    int result = binarySearch(array, 51, 0, SIZE-1);
    printf("%d\n", result);
}
```

之所以举二分查找这个例子，是因为这个递归函数的实现有些特别，它的基本条件不止一个。“if(low>high)”仅仅是所谓的基本条件之一。即当在局部范围内无法找到待搜索值时就表示搜索失败，递归过程也就结束了。此外，就是当找到待搜索值时，递归过程也会结束。这里的条件不是很明显，因为它采用了一种隐式的表达方式，即“else return mid;”，将这一句改写成“else if(a[mid]==x) return mid;”就明显多了。

2. 总是假设递归调用是有效的

在很多情况下，递归的过程可能很长，递归的分支也可能较多，很多初学者往往对此感

到困惑,有人难免会担心递归的过程是否能够按计划正确无误地进行下去。尤其在出现编译错误或逻辑错误时,更是感觉无从下手。但是无论在设计递归函数时如何设法跟踪很长的递归调用过程都是非常不明智而且徒劳的。于是有学者提出使用递归的另外一条原则是在设计递归算法时,如果递归的过程最终能够归结于基本条件,那么就总是假设递归调用是有效的。

 **提示:** 数学归纳法是递归的数学基础。递归的有效性就如同数学归纳法一样可以得到精确的保证。数学归纳法的基本思想就是在确定初始条件成立的情况下,试图证明一个递推关系的成立,如果初始条件成立,并且递推关系也成立的话,那么最开始的那个假设或等式就一定成立。

假设包含一个可能任意大的整数 N , 尽管我们没有办法一一验证所有可能的整数 N 对于等式都成立,但只要能够在初始条件成立的基础上论证递推关系的正确性,就大可不必关心 N 到底等于多少,也无须再考虑从 1 到 N 之间经过的所有具体步骤,这些因素对于结论的成立都没有影响。

递归的过程也是如此,程序员大可不必关心到每次递归的下一步递归是否能够正确进行,只要保证基本条件存在,并且所有递归过程的方向都向着基本条件进行即可。

3. 应尽量避免过度使用递归

递归可以使程序的风格明朗清晰,使程序的结构干净利落。但是递归并不是一种经济的做法。先来看一个例子——斐波那契数列的递归实现。

```
int Fib(int n){  
    if(n<=1) return n;  
    else return Fib(n-1)+Fib(n-2);  
}
```

图 5.12 所示为斐波那契数列的递归调用树。

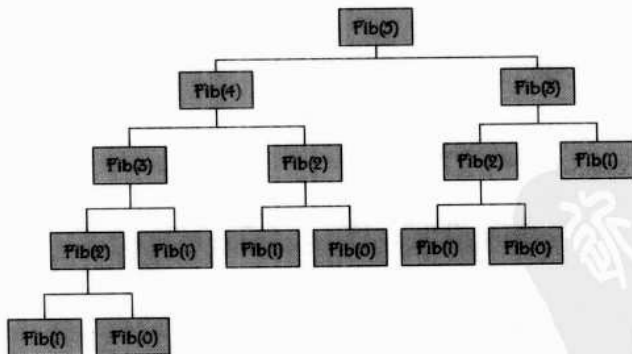


图5.12 斐波那契数列的递归调用树

分析可知求解斐波那契数列的调用次数 $\text{NumOfCall}(k) = 2 * \text{Fib}(k+1) - 1$, 这是一个比较大的开销。事实上,采用非递归的方式求解这个问题的计算量远小于使用递归的方式。一方面递归的过程需要更多的内部调用次数;另一方面,递归总是使用簿记的方式来跟踪前一个



调用块的位置，以保证递归后程序能够正确地返回调用点。而记录这些地址也需要消耗一定的时间，因此导致了效率下降。特别是在递归调用的过程非常长的情况下，对栈空间的开销也很惊人，而栈空间又是十分宝贵的。因此要避免使用过多的递归，尤其在效率要求高，而递归调用链过长的情况下。

在很多情况下，递归实现的函数都可以转换为非递归的形式。而且读者也应当意识到单纯为了追求形式上的清晰有时并不明智。因此很多地方尽管可以使用递归来实现，但在效率和形式之间往往还需要一些权衡。况且某些情况下使用递归明显是画蛇添足，尤其对于一些本来就属于非递归范畴的问题，如果非要将其转为递归描述，就显得有点不伦不类了。例如求前 n 个整数的和 $\text{Sum}(n)$ ，根据等差数列求和公式可以知道 $\text{Sum}(n) = n(n+1)/2$ ，当然用一个简单的累加循环也非常容易求解。下面这段代码则给出了一个递归的描述方式：

```
int Sum( int n )
{
    if ( n==1 )
        return 1;
    else
        return Sum(n-1)+n;
}
```

尽管它在功能上也实现了前 n 个整数求和的功能，但它显然不够高效，而且在形式上，它也谈不上清晰易懂，反而有点晦涩、古怪，因此在使用递归的时候一定要注意适度。

一些递归求解的问题完全可以转化为非递归方式求解，有时这种转化并不会太多地影响程序形式上的清晰，但在效率上却大有精进。例如下面这段代码，它使用了一种迭代的方式重写了二分查找算法：

```
#include "stdio.h"
#define SIZE 10

int binarySearch(int num[SIZE], int value)
{
    int low = 0, high = SIZE-1, mid;
    while(low < high)
    {
        mid = (low + high)/2;
        if(value < num[mid])
            high = mid -1;
        else if(value > num[mid])
            low = mid +1;
        else return mid;
    }
    return -1;
}

void main()
{
    int array[10] = {2, 3, 9, 10, 15, 27, 35, 50, 51, 99};
    int result = binarySearch(array, 51);
}
```

```
    printf("%d\n", result);  
}
```

上述二分查找算法的改写是非常成功的，它在得到效率上的提升之余，几乎并没有损失什么。可是，情况并非总是这样。例如下面给出求解最大公约数的欧几里德算法函数是基于递归实现的：

```
int gcd(int a, int b)  
{  
    if (b == 0)  
        return a;  
    else  
        return gcd(b, a%b);  
}
```

而下面这个求解函数则是用迭代算法来实现的：

```
int gcd ( int a, int b )  
{  
    while ( b != 0 )  
    {  
        int r = a % b;  
        a = b;  
        b = r;  
    } return a;  
}
```

很多迭代算法通常需要一个临时变量，有时这会导致程序的可读性降低，即使给定欧几里德算法的某些必要知识，试图通过简单的检查来理解其迭代求解过程也是比较困难的。但是，这对于求解最大公约数的欧几里德算法的执行，采用迭代算法在效率上的优势还是很明显的。

因为迭代函数不像递归函数那样需要考虑函数调用的支出，尤其当这种调用的数目巨大时，消耗也将是非常惊人的。另一个选择迭代而非递归的原因是，在当今的程序设计语言中，对于一个线程来说可用的栈空间通常比可用的堆空间要少得多，而递归算法则相对迭代算法需要更多的栈空间。因此出于这点的考虑，使用递归也不够经济。

当然，并非所有的递归函数都能够采用非递归的方式重写，这一点也是读者需要明确的。至于使用还是不使用递归，则应当完全依据具体问题而定。基本原则就是要尽量避免过度使用递归，千万不要形成某种思维定式，要注意分析具体的问题，在形式上和效率上求得一种平衡或取舍。毕竟，在程序设计的世界里往往都没有最理想的方式，有的只是最合适的方式。

4. 应好好理解递归调用的顺序

执行递归调用的顺序往往令人感到困惑，其实这并不是一个非常复杂的问题，但很多人直到一些奇怪的问题出现之前都忽略了它。首先请读者看下面这段示例程序：

```
#include "stdio.h"  
  
void func1(int a)  
{
```



```
    if (a < 4)
    {
        printf ("%d \n" , a);
        func1(a + 1);
    }
}

void func2(int a)
{
    if (a < 4)
    {
        func2(a + 1);
        printf ("%d \n" , a);
    }
}

void main()
{
    func1(0);
    func2(0);
}
```

编译并运行上述程序，输出结果如图 5.13 所示。

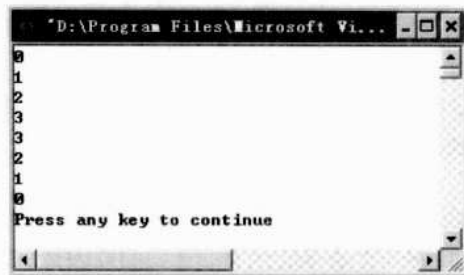


图5.13 递归函数运行结果

上述程序输出了奇怪的结果。变动调用函数的顺序竟然导致了函数执行顺序的变化。产生这样的结果是什么原因呢？可能很多读者还在被这个疑问所困扰。要解释产生这种现象的原因，就需要对递归过程和递归工作栈有所了解。递归的过程可以总结为以下几点：

- ☑ 递归过程的实现就是要自己调用自己。
- ☑ 递归调用的过程总是层层向下的，而退出时的次序则刚好相反。
- ☑ 主程序首次调用递归过程为外部调用。
- ☑ 递归过程每次递归调用自己都属于内部调用。
- ☑ 一个递归调用的函数，每次函数返回的地址都各不相同。

为了完成上述过程，系统需要开辟一些新的簿记空间来记录跟踪每一个递归调用，特别对于那些有一长串递归调用的情况，在某种程度上较同等循环更加费时，因为簿记工作本身就要消耗一定时间。这个簿记空间就是所谓的“递归工作栈”。函数在每一次递归调用时，此过程中所需要使用的参数、局部变量等信息都被存放在一块新开辟的空间中。每层递归调

用需分配的空间则形成了递归工作记录, 这些记录则按照后进先出的栈结构来进行组织。

基于这些原理, 我们就可以解释上述示例程序的执行过程了。对于函数 `func1`, 该函数首先执行了一次外部调用 `func1(0)`, 再进入函数块执行 `printf(0)`, 然后进行第一次内部调用 `func1(0+1)`, 进入函数块执行 `printf(1)`。如此继续下去, 最终调用了 `func1(3+1)`, 进入函数体后, 遇到判断条件 `num < 4` 时, 由于返回了 `false`, 因此不执行任何操作, 也就是默认执行了 `return`。返回到前一个调用块的下一条指令, 对于函数 `func1` 而言, 每一个调用块的下一条指令都是 `return`。整个过程如图 5.14 所示。

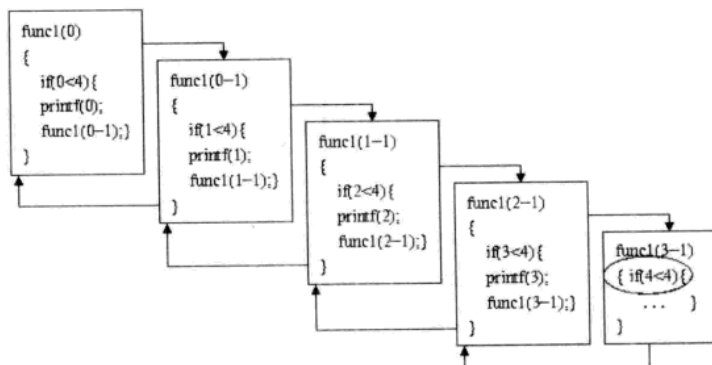


图5.14 函数func1执行过程

对于函数 `func2`, 该函数首先执行了一次外部调用 `func(0)`, 进入函数块后紧接着又进行了一次内部调用 `func(0+1)`, 然后依次进行了内部调用直到调用了 `func(3+1)`, 当遇到判断条件 `num < 4` 时, 由于返回了 `false`, 因此不执行任何操作, 也就是默认执行了 `return`。返回到前一个调用块的下一条指令也就是 `printf(3)`, 继续执行不断地返回到前一个调用块的下一条指令依次为 `printf(2)`、`printf(1)` 和 `printf(0)`。整个过程如图 5.15 所示。

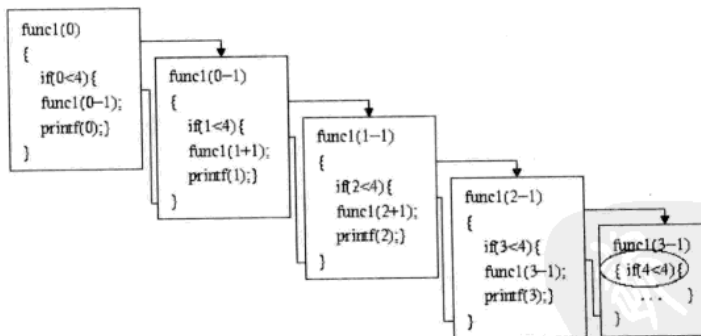


图5.15 函数func2执行过程

通过上述讲解, 不难发现, 变动递归调用函数的顺序有可能导致整个函数的执行顺序都发生变化。要想让函数的运行不至于超乎原有的设想, 那么一方面要求编程者在编写代码时小心谨慎地注意这些可能引发问题的地方。另一方面, 也要求编程者务必能够清醒明确地理解递归函数调用过程具体是如何实现的。只有对递归机制的根本原理有一个透彻的认识, 才能在使用递归的时候做到得心应手、事半功倍。



5.3.3 分治法与汉诺塔

1. 分治法

在电影《永不消逝的电波》中有这样一段情节：敌人发现有地下党的电报台在向外发送情报，但是敌人不能确定其准确位置。于是敌人尝试将城区分成两部分，即东城区和西城区。先将东城区停电，再看是否还有电台在工作，如果有则证明电台位于西城，否则即表明电台位于东城。再继续将东城或西城进而分成两个部分，如此继续下去最后就可以准确地发现电台的位置。

这是一种普通的一分为二处理问题的思想，这种思想也是分而治之思想的一种最简单形式。假想总是把问题一分为二地变成两个更小规模的子问题，并按照同样的程序一直分解下去直到问题小到能够轻而易举地解决，那么大问题也就随之而解了。这种将大问题逐渐分解，最终再分别求解的思想就是“分治法”。

2. 分治法的基本思想

分治法的基本思想是将一个规模较大的问题分解为若干个规模较小的子问题，且这些子问题相互独立并与原问题相同。这种思想古已有之，例如秦灭六国，统一天下正是采取各个击破、分而治之的原则。当一个问题规模较大不易求解的时候，就可以考虑将其分成几个小的模块，逐一解决。分治是算法设计中的一个重要思想，在解决具体问题时，反复使用分治手段，可以使原问题的规模不断缩小，直到子问题小到很容易解出，那么原来的问题也就迎刃而解了。由于分治法产生的子问题往往是原问题的较小模式，这就为使用递归提供了方便。

在应用分治思想设计具体算法时，需要考虑到应该把原问题分解为多少个子问题。从大量实践和经验角度来说，通常子问题的规模应当大致相同，也就是说把一个问题分成大小相等的若干个子问题的处理方式较为有效。在更多的情况下，问题总是被分为两个子问题，以期达到一种相对的平衡。这也就是我们在本节开始时所说的“二分法”。

3. 分治法求解“伪币问题”

下面来看一个可以利用分治法进行求解的“伪币问题”：假设一个袋子中装有 16 枚硬币。16 枚硬币中有且仅有一枚是伪造的，并且这枚伪币要较真的硬币轻些。现在考虑如何找出这枚伪造的硬币。这里仅提供一台可用于比较两组硬币重量的天平，通过这台仪器，可以知道两组硬币的重量是否相同。

☐ 我们思考一下如何使用分治法来解决这个问题。

假如把 16 枚硬币的例子看成一个大的问题。那么首先把这一问题分成两个小问题：随机选择 8 枚硬币作为第一组标记为 A 组，剩下的 8 枚硬币作为第二组标记为 B 组。然后用天平来比较 A 组硬币和 B 组硬币的重量。假如两组硬币重量不相等，则伪币存在于较轻的那一组硬币中。接着重复这一步骤，将较轻的一组再次分成两组，用天平比较后取其中较轻的一组。如此下去最多通过 4 次比较就能找出伪币。

☐ 再来看看如果用普通的方法来求解情况又当如何。

首先随意挑选两枚硬币，标记为 1 和 2。比较硬币 1 与硬币 2 的重量。假如硬币 1 比硬币 2 轻，则硬币 1 是伪造的；如果硬币 2 更轻，那么硬币 2 是伪造的。假如两硬币重量相等，则说明两枚硬币都是真的，继续比较硬币 3 和硬币 4。同样，假如有一个硬币轻一些，则寻

找伪币的任务完成, 否则继续比较剩余的硬币。按照这种方式, 最多通过 8 次比较即能找出这一枚伪币。

相比较而言, 采用分治法, 问题求解的计算量被大幅度降低。

4. 分治法进行求解(汉诺)塔问题

Hanoi (汉诺) 塔问题是利用分治法进行求解的一道经典数学问题, 它也是一个用递归方法解题的典型示例, 图 5.16 为汉诺塔模型。下面来分析一下 A 座上仅有 3 个盘子的时候, 如何将这 3 个盘子移动到 C 座上。



图5.16 汉诺塔模型

若想移动底部的 3 号盘子, 则必须移开它上面的 2 号盘子, 而若想移动 2 号盘子, 又必须移动上面的 1 号盘子, 如图 5.17 所示。

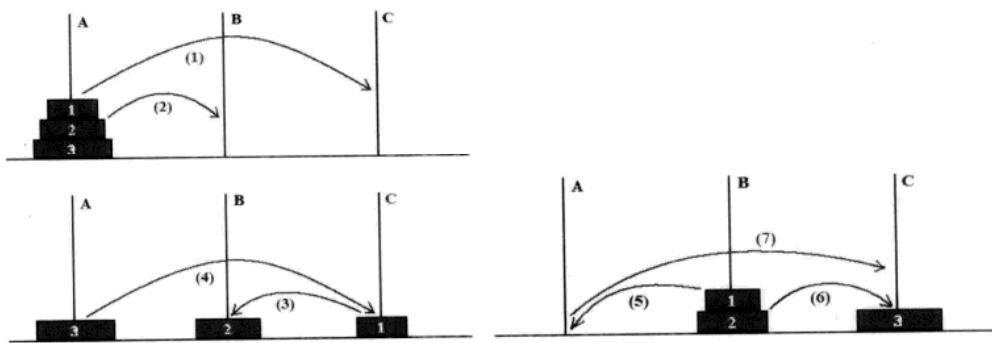


图5.17 三阶汉诺塔的求解过程

首先经过步骤(1)将 1 号盘子从 A 移到 C, 再经过步骤(2)将 2 号盘子从 A 移到 B。然后经过步骤(3)将 1 号盘子从 C 移到 B, 并经过步骤(4)将 3 号盘子从 A 移到 C。这样 3 号盘子就移动到 C 上了。再考虑将 2 号盘子移动到 C 上。所以经过步骤(5)将 1 号盘子从 B 移到 A, 再经过步骤(6)将 2 号盘子从 B 移到 C, 最后经过步骤(7)将 1 号盘子从 A 移到 C, 整个过程就完成了。当盘子数增加时, 只要按着这样的递归规则来移动, 最初的大问题就会逐渐被分解为规模更小的问题, 最终当小问题被逐个击破之后, 大问题也就随之解决了。

从上述简单的情况出发将问题推广, 可见当盘子的数目为 1 时, 只要将盘子从塔座 A 直接移动到 C 上即可。当盘子的数目 n 大于 1 时, 则需要利用塔座 B 来作为辅助塔座。这时需要想办法将 $n-1$ 个较小的圆盘依照规则从 A 移动到 B 上, 再将剩下的最大的盘子从 A 移动到 C, 最后, 再将 $n-1$ 个小盘依照规则从 B 移动到 C。如此下去, n 个圆盘的移动问题就可以分解为两次 $n-1$ 个圆盘的移动问题, 也就是分而治之。



由于游戏规则限定每次只能移动一个盘，且不允许大盘放在小盘上面，所以移动 64 个盘子无疑是一项非常浩大的工程。据估计，解决 64 层汉诺塔问题总共需要移动盘子的次数将超过 1.8×10^{19} 。这是一个天文数字，若每微秒可以计算（并不输出）一次移动，那么求得最终结果几乎也需要一百万年。因此这里也仅能找出问题的解决方法并解决较小 n 值时的汉诺塔问题，用普通计算机来解决 64 层的汉诺塔是不现实的。下面就给出用 C 语言编写的求解汉诺塔问题的示例代码清单：

```
#include <stdio.h>

void move(char i, char j)
{
    //输出移动过程
    printf("Move from %c to %c.\n", i, j);
}

//递归函数
void hanoi(int n, char a, char b, char c)
{
    if(n==1)move(a, c); //基本条件
    else
    {
        hanoi(n-1, a, c, b);
        move(a, c);
        hanoi(n-1, b, a, c);
    }
}

void main() {
    int num = 0;
    printf("请输入盘子的数量:\n");
    scanf("%d",&num);
    printf("移动这%d个盘子的步骤如下:\n", num);
    hanoi(num, 'A', 'B', 'C');
}
```

请读者完成编码后编译并运行程序。

5.3.4 回溯法与八皇后问题

1. 八皇后问题描述

19 世纪德国著名数学家高斯提出了著名的八皇后问题，该问题描述如下：在 8 行 8 列的国际象棋棋盘上摆放着 8 个皇后。若两个皇后位于同一行、同一列或同一对角线上，则称它们为互相攻击。在国际象棋中皇后是最强大的棋子，因为它的攻击范围最大，图 5.18 显示了一个皇后的攻击范围。

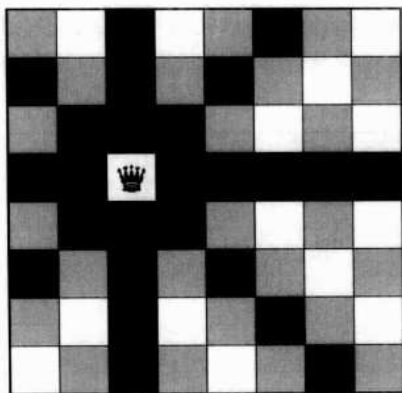


图5.18 皇后的攻击范围

现在要求使这 8 个皇后不能相互攻击,也就是说任意两个皇后都不能处于同一行、同一列或同一对角线上,问有多少种摆法。在现代教学中,把八皇后问题当做是一个经典的递归算法题目。图 5.19 显示了两种 8 个皇后不能相互攻击的情况。

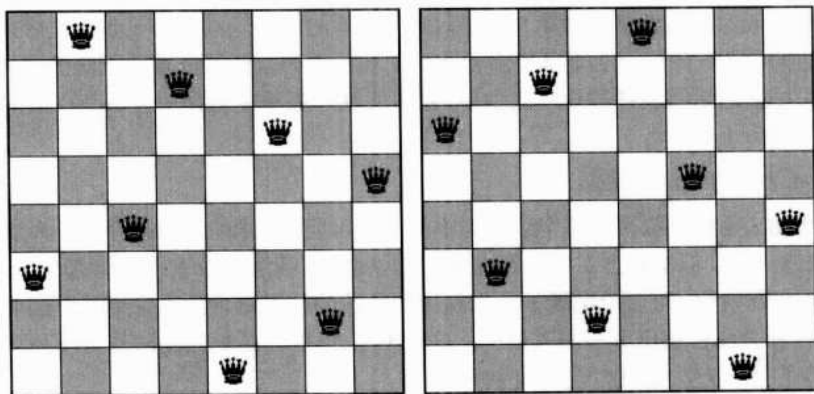


图5.19 8个皇后不能相互攻击的情况

2. 回溯法

八皇后问题是一个利用回溯思想进行求解的典型例题。回溯法有时也称为试探法,它是一种对问题可能的解空间进行系统性搜索的方法。回溯的过程是当某一种可能的试探结果否定了该可能路径的正确性后返回先前的某个状态继续进行其他可能性的试探过程。可以说回溯策略并非按照某种固定的计算方法来设计算法,而是通过尝试和纠正错误来寻找最终答案。

3. 使用回溯法解决八皇后问题

首先在棋盘上一列一列地摆放皇后,直到 8 个皇后在不相互攻击的情况下都被摆放在棋盘上,算法便终止。当一个新加入的皇后因为与已经存在的皇后之间相互攻击而不能被摆在棋盘上时,算法便发生回溯。如果发生这种情况,则试图把最后放在棋盘上的皇后移动到其他位置。这样做是为了让新加入的皇后能够在不与其他皇后相互攻击的情况下被摆放在棋盘



的适当位置上。例如图 5.20 所示的情况，尽管第 7 个皇后不会与已经放在棋盘上的任何一个皇后相互攻击，但仍然需要将它移除并发生回溯，因为第 8 个皇后在棋盘上无法找到合适的位置。

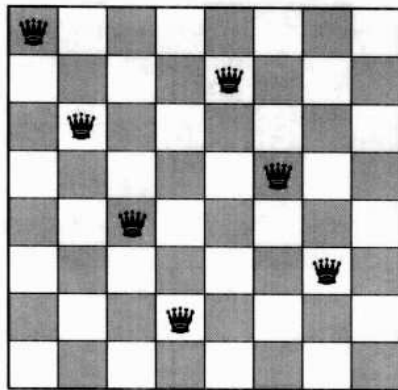


图5.20 需要发生回溯的情况

算法的回溯部分将尝试移动第 7 个皇后到第 7 列的另外一点来为第 8 个皇后在第 8 列寻找一个合适的位置。如果第 7 个皇后由于在第 7 列找不到合适的位置而无法被移动，那么算法就必须去掉它然后回溯到第 6 列的皇后。最终算法不断重复着摆放皇后和回溯的过程直到找到问题的解为止。

4. 回溯法与穷举法的区别

由于回溯法是在试图搜索整个解空间中的所有可能的选择，因此很容易让人误以为回溯法与穷举法差不多，但实际上二者是存在区别的。而且回溯法在效率上要比穷举法高出很多。仅以八皇后问题为例，可以粗略地估算出使用回溯法的计算量将约为使用穷举法的计算量的 2% 左右（估算方法已超出本书研究范围，这里不予详述），足见回溯法作为一种跳跃性和系统性相结合的搜索方法是具有较高效率的。

5. 求解八皇后问题的代码

下面给出求解八皇后问题的示例代码清单，需要说明的是由于该程序中用到了数组，而有关数组的内容被安排在下一章中进行介绍，因此如果读者对下述代码理解有障碍，则建议读者在完成数组部分的学习后再回过头来研究该程序：

```
#include <stdio.h>

int solution = 1;
int chess[8];

//根据前面几行的放子情况
//检查这一行放棋子是否合法
int attack(int n)
{
    int i;
    for (i = 1; i <= n - 1; i++)
```

```
{
    if (chess[n] == chess[i] + (n - i) ||
        chess[n] == chess[i] - (n - i) ||
        chess[n] == chess[i])
        return 0;
}
return 1;
}

//输出每种可行解
void display()
{
    int i;
    printf("Solution %d:\n", solution);
    for (i = 1; i <= 8; i++)
        printf(" %d", chess[i]);
    printf("\n");
    ++solution;
}

//递归求解函数
void putchess(int n)
{
    int i;
    //从第n行第一格(i)开始放棋子
    for (i = 1; i <= 8; i++)
    {
        chess[n] = i;
        if (attack(n) == 1)
        {
            if (n == 8)
                display();           //若已放满8行,则表示找出一种解
            else
                putchess(n + 1);      //若没放满则放下一行
        }
    }
}

int main()
{
    printf("八皇后问题的解如下: \n");
    //开始放置棋子
    putchess(1);
    return 0;
}
```

请读者完成编码后编译并运行程序。



5.4 变量的作用域

前面在介绍函数的参数传递时所给的示例程序告诉我们这样一个事实：即形参变量只能在函数调用时在函数内部被访问，当函数返回后，程序就无法再次使用其中的形参变量。也就是说，形参变量只有在函数内才有效，而当离开函数后将会失效。这种可以访问某个变量的代码区域就称之为变量的作用域。C 语言中所有的变量都有自己的作用域，变量的作用域表征了该变量有效的作用范围。而按照作用域的不同，C 语言中的变量可以分为局部变量和全局变量两种。

5.4.1 局部变量

局部变量就是指在一个函数内部定义的变量，因此它也称为内部变量，局部变量只在本函数内部有效，其作用域仅限于函数内，离开该函数后再使用这种变量就是错误的。请读者来看一段示例代码：

```
int f1(int a)
{
    int b = 0, c = 0;
    return a+b-c;
}

int f2()
{
    int x = 0;
    return x;
}

void main()
{
    int m, n = 0;
    m = f1(n)+f2();
}
```

在上述代码的主函数中声明了变量 `m` 和 `n`，它们都是局部变量，其作用域仅限于 `main()` 函数内部，尽管变量 `n` 被用作函数 `f1()` 的实参，但在函数 `f1()` 内部变量 `n` 是无效的。而在函数 `f1` 内也定义了 3 个变量，`a` 为形参，`b` 和 `c` 为一般变量。因此，`a`、`b` 和 `c` 变量的作用域仅限于 `f1` 内。同样，在函数 `f2()` 内定义的变量 `x`，也仅仅只在函数 `f2()` 内有效。

5.4.2 局部变量的作用域

关于局部变量的作用域有以下几点需要向读者说明：

- ☑ 形参变量是属于被调函数的局部变量，而实参变量则属于主调函数的局部变量。
- ☑ 允许在不同的函数中使用相同的变量名，但它们代表不同的对象，系统会把它们分开存放，它们彼此之间也不会发生混淆。例如我们把函数 `f2` 重写如下：

```
int f2()
{
```



```
int b = 0;
return b;
}
```

其中的变量 `b` 与函数 `f1()` 中的变量 `b` 是完全不同的两个变量。就相当于在美国有个人叫富兰克林，在英国也可以有个人叫富兰克林，但这两个富兰克林却是完全不同的两个人，彼此之间也不会互相影响。

☐ 主函数中定义的变量也只能在主函数中使用，不能在其他函数中使用。同样，在主函数中也不能使用其他函数中定义的变量。尽管主函数作为程序的入口而存在，但本质上所有函数之间的关系都是平行的。

☐ C 语言允许在复合语句中定义变量，其作用域只在复合语句范围内。来看下面一段示例代码：

```
#include "stdio.h"

void main()
{
    int x = 1, y = 0;
    //...
    {
        int sum;
        sum = x+y;
        //.....sum 的作用域
    }

    for(int i = 0; i<3; i++)
        printf("%d ", x); //这里需要注意，此句不能换为 printf("%d ", sum);
    for(; y < 2; y++)
    {
        char i = 'h';
        printf("%c ", i);
    }
    //...x 和 y 的作用域
}
```

上述代码中 `main()` 函数定义了两个局部变量 `x` 和 `y`，它们在整个 `main()` 函数中都有效。因此在后面的 `for` 循环中使用变量 `y` 是正确的。在变量 `x` 和 `y` 定义后紧接着的一对大括号中定义了一组复合语句，该复合语句中定义了一个局部变量 `sum`，该变量的作用域仅限于在这对大括号以内。

因此，如果将后面的语句 “`printf("%d ", x);`” 换为 “`printf("%d ", sum);`”，程序就会报错。第一个 `for` 循环的循环条件中定义的整型变量 `i` 的作用域仅限于该循环之中，因此在第二个 `for` 循环的循环体中又定义了一个字符型变量 `i` 是可以的。编译器并不会把两个 `i` 混淆，即使它们类型相同也无所谓，因为它们的作用域不同。

5.4.3 全局变量

相对于局部变量而言，在函数外部定义的变量就称为全局变量，全局变量也称为外部变量。全局变量不再属于哪一个函数，它的有效范围是从该变量定义的位置处开始直到源文件



的结尾。因此，在一个函数之前定义的全局变量，在该函数内无须说明即可使用。但是如果在该函数内定义了一个与之前定义的全局变量同名的变量，那么该同名局部变量就会在函数内部屏蔽全局变量的影响。例如下面这段示例代码：

```
#include "stdio.h"

int a,b;
int f1(int x)
{
    int y = x + 1;
    return y;
}

int c = 0, d = 0;
void f2()
{
    int d = 5;
    d++;
    printf("%d\n", d);
    //...
}

void main()
{
    int m = c;
    f2();
    printf("%d\n", f1(m+d));
    //...
}
```

此范围内全局变量 d 将被屏蔽掉

全局变量 a、b 的作用域

上例中 a、b、c、d 都是在函数外部定义的全局变量。但 c 和 d 定义在函数 f1() 之后，因此，它们在 f1 内无效。a 和 b 定义在源程序最前面，因此在函数 f1()、f2() 及 main() 内可以使用它们，但是在函数 f2() 内声明了一个局部变量 d，它将屏蔽掉全局变量 d，注意这两个变量是不同的。

使用全局变量最主要的一个作用就是允许多个函数都能对某个变量进行修改，这就意味着全局变量保持了一种方便多个函数有效沟通的渠道。例如，可以来看这样一个问题，假设有一个长方体的长、宽和高分别为 length、width 和 height。现在要求该长方体的体积及 3 个面的面积。


```
#include "stdio.h"

int s1, s2, s3;
int calc(int length, int width, int height)
{
    s1 = length * width;
    s2 = width * height;
    s3 = length * height;
    return length * width * height;
}
```

```
void main()
{
    int length = 30, width = 40, height = 50;
    int v = calc(length, width, height);
    printf("v=%d s1=%d s2=%d s3=%d\n", v, s1, s2, s3);
}
```

上述程序中函数 calc 将完成长方体 3 个面的面积及其体积这 4 个结果的计算功能。但是由于函数仅能有一个返回值，因此，程序中定义了 3 个外部变量 s1、s2 和 s3，它们分别用来存放 3 个面积的计算结果。

这些全局变量的作用域为整个程序，因此函数 calc 可以对其进行赋值，即使在函数调用结束后，它们的计算结果也仍然可以被保留。然后 main() 函数将结果输出，这其实就完成 main() 函数与被调函数之间的 3 次信息传递。函数 calc 再将体积计算结果以返回值的形式传回。main() 函数最后负责将 3 个面积和体积的计算结果输出。可见，外部变量是实现函数之间数据交流的有效手段。

 **注意：**尽管全局变量能够方便多函数之间的信息传递，但是我们仍然建议在非必要时尽量不要使用全局变量，原因有以下两点：

- ☑ 外部变量破坏了函数的独立性，函数不再以独立的形式来完成各自的功能，因为这些功能的实现将受制于外部因素。这与结构化程序设计的思想是相违背的。
- ☑ 全局变量在程序执行的过程中始终占用存储单元，而不像局部变量那样仅在使用时才被分配存储单元，全局变量必须要等到整个程序结束后才被释放，这也是一个不太经济的做法，因此在不必要时尽量不要使用全局变量。

5.5 变量的存储类型

变量作用域的不同归根结底是由于变量存储类型的不同而造成的。所谓存储类型就是指变量占用内存空间的方式，它也被称为“存储方式”。计算机中变量的存储方式可以分为：

- ☑ 静态存储。
- ☑ 动态存储。

静态存储的变量通常是在变量定义时就为其分配固定的存储单元并一直保持不变，直至整个程序结束，前面所说的全局变量就属于这种存储方式。而动态存储则更为常见，采用动态存储方式的变量在程序的执行过程中，仅当需要时才临时性地分配存储单元，并且在使用完毕之后，动态存储的变量也会被立即释放。

例如前面所介绍的局部变量，在函数定义时函数中的局部变量并没有被分配存储空间，只有在函数被调用时，局部变量才会被分配存储空间，当函数调用完毕后，其中的局部变量就立即被释放。如果一个函数被多次调用，那么其中的局部变量就会被反复地分配及释放存储空间。

总之，静态存储变量是一直存在的，而动态存储变量则是临时存在的。这种由于变量存



存储方式不同而产生的特性称为变量的生存周期。生存周期表示了变量有效的时间阶段。生存周期和作用域分别从时间和空间这两个不同的维度对变量的存在特性进行了描述。

要想辨明变量采用的是哪种存储方式，不能仅从作用域的角度来考察，更重要的是明确变量究竟采用了哪种存储方式。C 语言通过 `auto`、`static`、`register` 和 `extern` 这 4 个关键字对变量的存储类型进行说明，本节就向读者介绍和这 4 个关键字有关的一些知识。

5.5.1 auto 变量

1. 自动 (auto) 变量介绍

在程序开发实践中，用到最多的就是自动变量。自动变量采用的是动态存储方式，也就是说仅当函数被调用时其中的自动变量才会被分配存储空间，而且当函数返回后这些临时存储空间就会被释放。

C 语言中规定，函数内凡未加存储类型说明的变量均视为自动变量，所以实际上关键字 “`auto`” 是可以省略的，说明符 `auto` 不写则隐含表示该变量为自动存储类型。例如，在本书前面各章示例程序中所定义的未加存储类型说明符的局部变量和形参变量都是自动变量，例如：

```
int function(int a)
{
    auto int i;
    auto char c;
    //...
}
```

其中形参变量 `a` 以及函数内部定义的 `i` 和 `c` 就都是自动变量。而且上述程序片段等价于：

```
int function(int a)
{
    int i;
    char c;
    //...
}
```

2. 自动变量的作用域及生命周期

自动变量的作用域仅限于定义该变量的函数或者复合语句内，这一点在介绍局部变量时已经讲过。此外，由于自动变量属于动态存储方式的变量，因此它的使用寿命仅限于函数调用的过程中，当函数返回后，其生命周期也随之结束。因此函数调用结束之后，自动变量的值不能保留。

在复合语句中定义的自动变量，在退出复合语句后也不能再使用。也正是因为自动变量的作用域和生命周期都局限于定义它的函数（或复合语句）内，因此不同的函数（或复合语句）中可以使用同名的变量，并且不会发生混淆。即使在函数内定义的自动变量，也可与该函数内部的复合语句中定义的自动变量同名。只不过函数内部复合语句中定义的自动变量将屏蔽函数中本来定义的同名局部变量。

5.5.2 static 局部变量

1. 全局变量与静态 (static) 局部变量

前面已经说过静态存储方式是指在程序运行期间分配固定的存储空间的方式。全局变量采用的是静态存储方式。如果在函数外部定义一个变量，那么它就是一个全局变量。整个程序中的所有函数都可以使用这个全局变量，而且全局变量也会将每次的修改结果保留下来。但通常函数中的局部变量无法做到这一点，因为局部变量总是在函数返回后被释放。

有些时候，程序员可能希望函数中的局部变量在函数调用结束后不消失而保留原值，也就是希望占用的存储单元不被释放，而在下一次该函数调用时，该变量已有值，即上一次函数调用结束时的值。这时就应该指定该局部变量为“静态局部变量”，C 语言中使用关键词“static”来指示一个变量为静态局部变量。一个变量如果被声明为 static，那么它就会以静态存储方式来被分配存储空间。

2. 静态局部变量示例

请读者来看下面这个关于静态局部变量使用的例子。

```
#include "stdio.h"

void f()
{
    static int num = 4;
    printf("%d ", num);
    num++;
}

void main()
{
    for(int i = 0; i < 3; i++)
        f();
    printf("\n");
}
```

完成编码后编译并运行上述程序，程序的输出结果为：

4 5 6

但是如果将关键字“static”去掉，那么程序的输出结果就会变为：

4 4 4

这段示例程序的运行结果可以带给我们许多有用的信息：

- ☑ 静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间都不释放。

上例中，变量 num 的每次自加结果都得以保存，就是因为在程序的整个运行期间，其对应的地址都没有被释放。

- ☑ 静态局部变量是在编译时赋值的，即只赋初值一次。

在上例中尽管每次调用函数 f()，函数体中的第一条语句貌似都会为变量 num 赋值，然而事实并非如此。因为变量 num 是一个静态局部变量，所以它仅会被赋初值一次，而不会



被多次赋初值。所以，语句“static int num = 4;”事实上只执行了一次。

- ☑ 如果定义局部变量时不赋初值，则对静态局部变量来说，编译时会自动赋初值 0（对数值型变量）或空字符（对字符变量）。

这一点读者可以自己编程测试一下。如果局部变量并非是静态的，且它没有被指定初值，那么它的初值可能是任意数。

- ☑ 虽然静态局部变量在函数调用结束后仍然存在，但其他函数是不能引用它的，这一点与全局变量不同。

例如下面这段示例程序：

```
#include "stdio.h"

void f1()
{
    static int num = 4;
    printf("%d ", num);
    num++;
}

void f2()
{
    printf("%d ", num);
}

void main()
{
    for(int i = 0; i < 3; i++)
        f2();
    printf("\n");
}
```

在 Visual C++ 6.0 中编译上述程序，编译器将会抛出一个错误“error C2065: 'num' : undeclared identifier”。这是因为函数 f2() 访问了函数 f1() 中的静态局部变量 num，尽管函数 f1() 中的变量 num 被声明为一个 static 变量，但其他函数也是不能访问它的。

- ✎ **提示：**如果函数中的一个变量被声明为静态，那么这个变量就会被静态地分配存储空间。它只在函数内可见，但是变量不会在每次函数调用时都被创建。如果希望一个变量的值能够在修改后得以保存，并在程序运行期间内被某个可能反复调用的函数持续访问，又希望对其他函数隐藏该变量，那么静态局部变量是一个好的选择。

5.5.3 register 变量

1. 变量、内存与寄存器

变量在一般情况下都是存储在内存中的，计算机中的内存又称为 RAM（随机访问存储器），它是一种半导体存储器件。如果程序需要使用某个变量，CPU 就会从内存中取值，然后做进一步处理。CPU 从内存中取得变量值后会将其暂时存放在寄存器中。

寄存器就是 CPU 自己的“小内存”，如图 5.21 所示，其特点是“容量小，速度快”。在


正常情况下, 编程语言本身是无法直接操作寄存器的。但在某些时候, 一些变量有可能会被频繁地使用到。频繁地对内存进行存取操作就有可能耗用较多的时间, 如果我们能够有效地将 CPU 的寄存器利用起来, 就会有效地提升程序的运行效率。



图5.21 寄存器访问

2. 寄存器变量

C 语言中提供了一种帮助程序员利用 CPU 寄存器的语法。使用关键字 `register` 来声明局部变量时, 该变量即称为寄存器变量。寄存器变量的值会被存放在 CPU 寄存器中, 每当需要使用它们的时候, CPU 就可以直接使用, 而无须再从内存中获取。

 **注意:** CPU 访问寄存器的速度要远远快于其访问内存的速度, 所以正确地使用寄存器变量能够有效地提高程序运行效率。

下面给出了一段使用寄存器变量的示例程序:

```
#include "stdio.h"

unsigned long int sum(int n)
{
    register unsigned long int s = 0;
    register int i;
    for(i = 1; i <= n; i++)
        s += i;
    return s;
}

void main()
{
    unsigned long int result = sum(2000);
    printf("sum(2000) = %d\n", result);
}
```

上述程序中的函数 `sum()` 实现了一个简单的求和功能, 由于在求和过程中局部变量 `i` 和 `s` 要被频繁使用, 于是声明它们为寄存器变量, 这样一来就可以提成程序的整体运行效率。

3. 使用寄存器变量的注意事项

❑ 不能定义任意多个寄存器变量。

尽管寄存器变量的访问速度远高于普通变量的访问速度, 但是一台计算机中的寄存器数量是有限的, 甚至可以说是非常匮乏的。而且对于不同的系统来说, 所允许使用的最大寄存器数量也是不同的。在 Intel 体系中 CPU 通常所具有的寄存器个数大约在 6~16 个之间。不同系统上的寄存器数量可能不同, 所以在编程时所允许使用的寄存器数量也会不同。但寄存器资源相对来说总是稀缺的, 所以滥用也会造成浪费。



☐ 不同系统对于寄存器变量的处理方式可能不同。

有的系统只允许将 `int`、`char` 和指针型变量定义为寄存器变量，而有的系统则将寄存器变量当做是自动变量来看，并不真正地把它存放在寄存器中。这一点也希望读者能够明确。

☐ 只有局部自动变量和形式参数才能够被定义为寄存器变量。全局变量和局部静态变量都不能被定义为寄存器变量。

全局变量和局部静态变量都是被放在静态存储区中的，而寄存器变量是被放在寄存器中的，一个变量当然只能选择两种存放方式中的一种。所以，全局寄存器变量或者静态寄存器变量都是不存在的。

事实上，经过多年的发展与严谨，现代编译器已经非常强大，大部分商用编译器都可以非常智能地进行许多优化工作。目前，这些编译器基本上可以识别出那些会被频繁使用的变量，对于这些变量编译器会把它存放在寄存器中，而不需要程序员在代码中显式指定，所以寄存器变量的声明在某种程度上来说并不是必需的，读者仅对此有所了解即可。

5.5.4 extern 变量

1. 关键字 `extern` 在源文件中扩展变量的作用域

关键字 `extern` 可以用来修饰外部变量，这时它所起到的作用是扩展外部变量的作用域。前面已经讲过一个普通的外部变量的作用域是从其定义处起始直到源文件的结束，但是没有一种方法能够在定义某外部变量的语句出现之前就允许使用它。在 C 语言中允许此种情况的存在。其处理方式就是在变量被引用之前使用关键字 `extern` 来对其进行“外部变量声明”。下面这段示例代码演示了此种语法：

```
#include "stdio.h"

int flower()
{
    extern int n;    //外部变量声明
    int i, j, k;
    i = n/100; j = n/10-i*10; k = n%10;
    if(i*100+j*10+k == i*i*i + j*j*j + k*k*k)
        return n;
    return 0;
}


int n;

void main()
{
    int k;
    for(n = 100; n < 1000; n++)
        if(k = flower())
            printf("%d ", k);
}
```

上面这段程序的作用是求三位的水仙花数，并输出结果。水仙花数是指一个 n 位数 ($n \geq 3$)，它的每个位上的数字的 n 次幂之和等于它本身，例如 $1^3 + 5^3 + 3^3 = 153$ 。全局变量 `n`

用以完成主调函数和被调函数之间的信息传递。

这里举这个例子仅仅是为了说明关键字 `extern` 的一种最主要的用法。从上述代码中可以清晰地看到由于外部变量 `n` 定义在了函数 `flower()` 之后，因此它的原作用域应该不包括 `flower()` 函数。但是 `flower()` 函数中又要使用到该外部变量，于是就采用外部变量声明的方式来将外部变量 `n` 的作用域扩展。

 **提示：**用 `extern` 声明外部变量时，类型标识符是可以省略的，例如上例中的语句“`extern int n;`”写作“`extern n;`”也是正确的。

2. 关键字 `extern` 实现变量的跨文件访问

当一个程序的功能比较复杂时，那么它的源文件可能不止一个，也就是说不同的功能被分散地写到不同的源文件中。一旦出现这种情况，若是想让在一个源文件中定义的外部变量在其他的源文件中也有效，这时就可以考虑使用关键字 `extern` 来进行外部变量声明。例如，有一个源程序由两个源文件 `file1.c` 和 `file2.c` 组成，其中 `file1.c` 的内容如下：

```
int a,b;          /*外部变量定义*/
void main()
{
    //...
}
```

`file2.c` 的内容如下：

```
extern int a,b; /*外部变量说明*/
//...
func(int x)
{
    x = a + b;
    //...
}
```

由此可见，在 `file1.c` 和 `file2.c` 两个源文件中都要使用 `a` 和 `b` 两个变量。在 `file1.c` 文件中把 `a` 和 `b` 都定义为外部变量。在 `file2.c` 文件中用 `extern` 把这两个变量说明为外部变量，表示这些变量已在其他文件中被定义过。

3. 使用关键字 `extern` 需谨慎

如果误用了这个关键字，那么不同文件或函数中的值就会彼此产生连带效应。也就是说当一个文件中的某个外部变量被修改时，另外一个文件中的变量也会被更改。

无论这种更改是不是你所希望的，另一个文件中函数的执行结果都可能会因为其他文件中函数的执行而受到影响。一旦这种影响是错误的，那么这种错误就是非常隐蔽的，因此也增加了调试的难度。

4. 编译器对关键字 `extern` 的处理

关键字 `extern` 既能够在本文件范围内扩展外部变量的作用域，又可以将外部变量作用域扩展到其他源文件中，但是编译器在编译源程序的时候并不会搞混。因为在编译的时候，编译器总是从使用 `extern` 进行外部变量声明的地方开始在本文件中查找外部变量的定义。如图 5.22 所示，如果外部变量的定义在本文件中被找到，那么外部变量的作用域就在本文件中进



行扩展。如果在本文中无法找到外部变量的定义，那么编译器就会从工程中的其他文件中查找外部变量定义。这时变量的作用域就扩展到其他文件中。但是如果还是没有找到，编译器就会抛出编译错误。

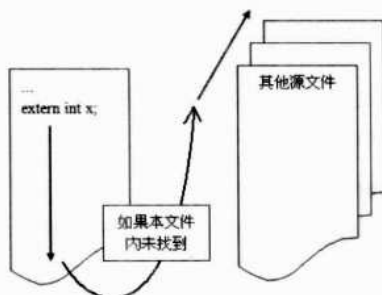


图5.22 extern的作用机制解析

5.5.5 static 外部变量

通过上一小节的学习，我们已经知道关键字 `extern` 会把外部变量的作用域扩展到其他文件中。但是有时在程序设计中，开发人员可能会希望某些外部变量只限于在本文件中被引用。这种限制条件提高了程序的健壮性和安全性，毕竟一个文件中的变量被修改后，其他文件的执行也会受到影响，从而会埋下一些不安全的隐患。为了克服这种隐患，可以在定义外部变量时加一个 `static` 声明。下面这段示例代码演示了这种用法，假设工程中有两个文件，其中文件 `file1.c` 的内容如下：

```
static int a,b;    /*静态外部变量*/
void main()
{
    //...
}
```

`file2.c` 的内容如下：

```
extern int a,b;    /*外部变量说明*/
//...
func(int x)
{
    x = a + b;  /*!!!错误*/
    //...
}
```

在 `file2.c` 文件中，声明了两个外部变量 `a` 和 `b`，并在函数 `func()` 中使用了它们。因此编译器在编译程序时就会到其他文件中去找，结果发现在 `file1.c` 文件中两个变量被 `static` 所修饰，因此不能使用，于是程序就会抛出编译错误。

在定义外部变量时，如果使用 `static` 来修饰变量，那么这个外部变量就仅能在此文件中使用，这是非常有用的。因为在实际开发中，一个工程中的源文件不止一个，而且参与开发的人也有很多。不同的人所写的不同文件中，很可能有同名的外部变量。使用 `static` 关键字就可以让这些同名变量彼此之间相互区分，消除连带影响。

5.6 执行多文件程序


实际的开发项目其功能模块的划分是相当复杂的。不同的模块将会被写入不同的源文件中,并由不同的程序员来分别完成。不同的文件之间要共同构成一个有机的程序,那么文件与文件之间也需要交流。除了依靠外部变量以外,函数的相互调用也必不可少。本节就向读者介绍多文件程序中函数的调用方法,以及如何运行一个多文件程序。

5.6.1 内部函数

变量有作用域,函数也有。尽管函数本质上是全局的,但可以限定函数能否被别的文件所引用。当一个源程序由多个源文件组成时,C语言根据函数能否被其他源文件中的函数调用,将函数分为内部函数和外部函数。

如果一个函数只能被本文件中其他函数所调用,则称为内部函数。内部函数又称为静态函数。在定义内部函数时需要在函数名和函数类型前面加 `static` 关键字,即内部函数的一般定义形式如下:

```
static 类型标示符 函数名(形参表)
{
    //函数体
}
```

 **注意:** 与静态变量不同,这里的 `static` 含义不是指存储方式,而是指对函数的作用域仅局限于本文件。

内部函数的使用会增加函数的访问限制,这增强了程序的健壮性和安全性。特别在一个具有一定规模的实际项目中,不同的人编写不同的函数时,大家都不必担心自己定义的函数是否会与其他文件中的函数同名。这时,添加一定的限制是非常有必要的。

5.6.2 外部函数

如果一个函数可以被其他文件中的其他函数所调用,那么它就是一个外部函数。在定义外部函数时需要在函数名和函数类型前面加关键字 `extern`,也可以省略不写,这就表示此函数是一个外部函数,即外部函数的一般定义形式如下:

```
extern 类型标示符 函数名(形参表) //extern 亦可省略不写
{
    //函数体
}
```

只要用关键字 `extern` 来修饰函数,那么该函数就可以被其他文件中的函数所调用。例如,下面这个示例程序中共包含 4 个源文件:

- ☑ `mainfile.c`。
- ☑ `output.c`。
- ☑ `input.c`。
- ☑ `process.c`。

这个程序的作用是接收一个字符串,然后将该字符串反转,并输出结果。该程序的主函



数位于文件 mainfile.c 中，且主函数调用了函数 getString()、output() 和 reverse()，而这 3 个函数又分别位于文件 input.c、output.c 和 process.c 中。为了能够让主函数成功地调用它们，我们使用了关键字 extern。mainfile.c 文件的代码清单如下：

```
#include "stdio.h"
#include "string.h"

int main()
{
    extern void getString(char str[]);
    extern void output(char str[]);
    extern void reverse(char str[], int low, int high);

    char text[50];

    printf("请输入字符串，不要超过 50 个字符:\n");
    getString(text);

    reverse(text, 0, strlen(text)-1);
    printf("反转后的字符串为:\n");
    output(text);

    return 1;
}
```

input.c 文件的代码清单如下：

```
#include "stdio.h"
//获取字符串
void getString(char str[])
{
    gets(str);
}
```

output.c 文件的代码清单如下：

```
#include "stdio.h"
//输出字符串
void output(char str[])
{
    printf("%s\n", str);
}
```

process.c 文件的代码清单如下，函数 reverse() 用于实现字符串的反转，可见它是采用了递归的方式来实现的：


```
//字符串反转处理函数
void reverse(char s[], int l, int h)
{
    if(l > h) return;
    else
    {
        char t;
```

```

reverse(s, l+1, h-1);
t = s[l], s[l] = s[h], s[h] = t;
}
}

```

完成编码后, 请读者编译并运行上述程序。

 **提示:** 关于字符串使用方面的更多内容被安排在了本书的第6章, 如果读者觉得相关语法在理解上有障碍, 那么建议在完成第6章相关部分的学习后再来仔细研究本程序。

5.6.3 多文件程序实例

1. Visual C++ 6.0 执行多文件程序的步骤

不同编译器下运行程序的方法各有区别, 但整体上仍然保持一致。下面就以 Visual C++ 6.0 为例来向读者介绍执行一个多文件程序。

第1步 新建一个项目, 然后向其中创建新的源文件, 如图 5.23 所示。

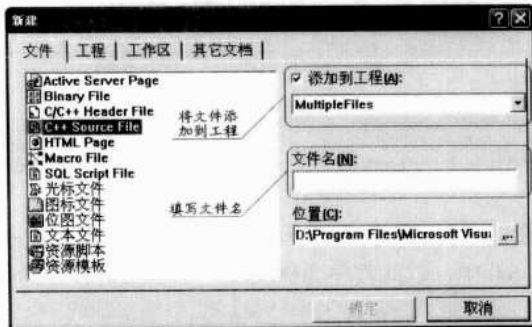




图5.23 添加新文件

从左侧的列表框中选择要创建的文件类型, 然后在“文件名”文本框中填写要创建的文件名。填写完文件名后还需要勾选“添加到工程”复选框, 并从其下拉列表中选择目标工程。

 **注意:** 不要使用项目中已经存在的文件名, 另外此处也无须添加扩展名。

第2步 分别编辑各源文件, 编码完成后即可正常进入编译、连接和运行步骤。

 **注意:** 与单个源文件相同, 此时编译产生的目标文件, 以及连接产生的可执行文件, 它们的主文件名均与项目文件的主文件名相同。

2. 如何调用其他文件中的函数

当程序中某个文件里调用了另外一个文件中的函数时, 除了使用关键字 `extern` 以外, 还可以直接引用文件。如图 5.24 所示, Visual C++ 6.0 界面中左边的文件浏览栏中列出了此项目中的所有文件。当需要引用文件时就使用语句:

```
#include "文件名"
```

其中文件名可以是.c 类型的源文件, 也可以是.h 类型的头文件, 例如本例中也可以将 `input.c`、`output.c` 和 `process.c` 个文件改为 `input.h`、`output.h` 和 `process.h`。



提示：在 Visual C++ 6.0 中，更多地使用 .cpp 类型文件代替 .c 类型的文件，.cpp 类型的文件是属于 C++ 语言的源文件类型，因为 C++ 包含了 C 语言，因此 C 语言的源文件在 C++ 编译器中同样可以编译运行。

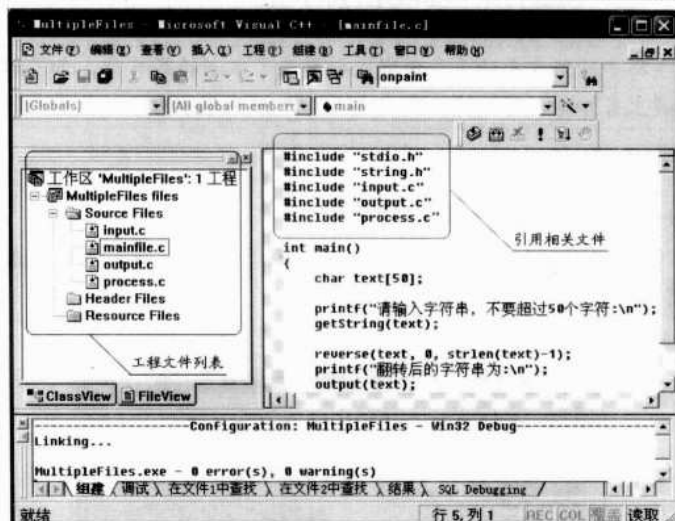


图5.24 项目中的文件

以 5.6.2 小节中给出的字符串反转程序为例，下面给出改写后的 mainfile.c 文件的代码清单，读者在 Visual C++ 6.0 中，以此程序为例实践一下多文件程序的执行过程。

```
#include "stdio.h"
#include "string.h"
#include "input.c"
#include "output.c"
#include "process.c"

int main()
{
    char text[50];

    printf("请输入字符串，不要超过 50 个字符:\n");
    getString(text);

    reverse(text, 0, strlen(text)-1);
    printf("反转后的字符串为:\n");
    output(text);

    return 1;
}
```



第 6 章

数组与字符串



英语谚语中讲：长着同样羽毛的鸟儿会飞到一起。这也类似于中国俗语所说的：物以类聚、人以群分。现实中的事物往往都不是孤立存在的，数据也一样，它们更倾向于重组成一个集合。为了方便现实中许多问题的处理，人们引入了数组的概念。使用这种最简单的结构化数据类型能够使程序在解决许多复杂问题上游刃有余。





6.1 一维数组的使用

一维数组是最简单的数组，它的应用也最广泛，本节就从一维数组的概念入手，介绍它的基本使用方法。

6.1.1 数组概念的引入——中国古代军队编制

1. 现实版数组表现

根据《周礼》记载，中国古代军队里“五人为伍，五伍为两，五两为卒，五卒为旅，五旅为师，五师为军。”从西周时代起，就是按伍、两、卒、旅、师、军编制排列的。后来，商鞅变法，建立秦国新军连保制：“五人一伍，头目称伍长；十人一什，头目称什长；五十人为一属，头目称属长；百人一闾，头目为闾长，俗称百夫长……”东汉军制又为：“五人一伍，有伍长；十人一什，有什长；五十人一队，有队长；百人一屯，有屯长；二百人一曲，有军侯；千人一部，有司马或校尉为正司马为副。”这也就后来“队伍”一词的出处。

这种编制行为揭示了现实世界中，将同种类型数据进行有序组织以便于管理的思想，从而引出了数组的定义——数组是一种简单而常用的线性数据结构，简单地说就是相同类型数据项的集合。

2. C 语言中数组的定义

相对于整型、浮点型和字符型这些基本数据类型而言，数组较它们有一个重大的飞跃。数组是最简单、最基本的结构型数据类型，也就是说数组是由原子数据按一定顺序关系存放在一起的一个数据集合，组成数组的对象称为该数组的元素。回过头来再看看东汉的军队编制方式，可知“五人一伍”，那么也就是说这个数组的容量是 5，数组中的数据项是“人”。

C 语言中一维数组的定义方式如下：

类型说明符 数组名[常量表达式]；

例如，下面几个都是正确的数组定义：

```
int array[10];
double num5[5];
char name_person[20];
```

数组定义中的类型说明符用来说明数组中元素的类型。而数组名则是与数组对应的一个标识符，数组的命名规则与变量名相同。数组名后面用方括号括起来的常量表达式用来表示数组中元素的个数，例如 `int array[10]` 就表示名为 `array` 的数组中有 10 个 `int` 类型的变量。

另外常量表达式中可以包含常量和符号常量，也可以是一个用宏定义来表示常数值（关于宏定义的更多内容会在本书后面介绍）。但是数组定义的常量表达式中不能含有变量，这是因为数组一旦被定义，它的大小就已经被划定，无法动态地改变，所以数组的长度也就不能是一个变量了。例如，下面的数组定义就是错误的：

```
int n;
scanf("%d", &n);
int array[n];
```

6.1.2 数组元素的使用

数组在使用之前必须先定义，这一点与普通变量无异。但是数组毕竟是多个元素形成的集合，因此在实际使用的时候我们所面对的往往都是其中的具体元素项，而非整个数组。在引用具体数组元素的时候，就需要用到数组的下标。数组下标就是指在数组名后的方括号内的数值或表达式，它可以用于指示访问哪个数组元素。例如定义一个含有 8 个元素项目的整型数组 *a* 如下：

```
int a[8];
```

那么在引用数组中的具体元素时就可通过下标来实现目的，例如若是希望把数组中的第 1 个元素置为 0，就可以使用下述语句：

```
a[0]=1;
```

像 *a[0]* 这种用来指示一个数组元素的变量就称为一个下标变量，它是以在数组名后面跟一个方括号（方括号内是数组的下标）的形式来表示的。这似乎看起来和数组的定义有点像，但是，读者一定要注意数组下标是从 0 开始计算的，也就是说数组 *a* 中的 8 个元素分别可以表示为：*a[0]*、*a[1]*、*a[2]*、...、*a[7]*。

并不存在 *a[10]* 这个元素，当对 *a[10]* 进行操作的时候将发生越界访问，这一点在后面还会详细介绍。图 6.1 演示了数组 *a* 中各个元素及其引用方式的对应关系，可见数组的第一个元素是 *a[0]* 而非 *a[1]*。

34	25	18	99	5	18	60	1
<i>a[0]</i>	<i>a[1]</i>	<i>a[2]</i>	<i>a[3]</i>	<i>a[4]</i>	<i>a[5]</i>	<i>a[6]</i>	<i>a[7]</i>

图6.1 数组

前面已经提过数组下标可以是数值，也可以是表达式，因此下面几个都是合法的数组元素引用方式：

```
int i = 3;           //声明一个整型变量 i
printf("%d", a[2]);  //输出数组中第 3 个元素的值
printf("%d", a[i]);  //输出数组中第 4 个元素的值
printf("%d", a[i+2]); //输出数组中第 6 个元素的值
printf("%d", a[i*2]); //输出数组中第 7 个元素的值
printf("%d", a[i++]); //输出数组中第 4 个元素的值，然后将 4 赋给 i
printf("%d", a[--i]); //将 3 (=4-1) 赋给 i，然后将 a[3] 的值输出
a[i] = a[i+1]+a[i+2]; //将 a[4]+ a[5] 的值赋给 a[3]
```

6.1.3 数组的初始化

1. 数组中未赋值元素的值

在介绍具体内容之前，请读者先来看一段示例程序：

```
#include <stdio.h>

void main()
{
    int a[10];
```



```
printf("%d\n", a[5]);  
}
```

这段程序首先定义了一个数组 `a`，然后输出了数组中的一个元素 `a[5]`。问题随之而来，我们并没有给数组中的元素赋值，那么 `a[5]` 到底应该等于多少呢？图 6.2 给出了该程序的运行结果。不可思议的是程序输出了一个莫名其妙的值。

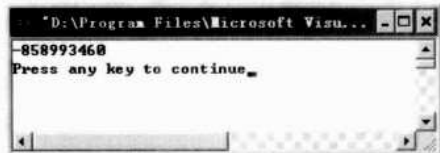


图6.2 程序运行结果

2. 莫名其妙的值的来源

编译器在数组定义的时候就开辟了一定的存储空间，在程序员对数组进行赋值之前，编译器也已经为它们赋予了初始值。但是这个初始值远远在我们的控制范围之外，我们根本就不知道编译器会给数组赋一个怎样的初值。

于是接下来程序的运行也就不在控制范围之内了。这是非常危险的，程序开发者应该能够准确地操纵程序，理解程序的运行，而不应该任由程序自生自灭。因此，我们得到了一个宝贵的编程经验，或者叫可取的编程习惯，即在使用数组之前最好对其进行初始化。

3. 数组进行初始化的方法

☐ 在定义数组时就给数组中的每个元素赋初值，可以将数组中元素的值用一个大括号括起来，每个元素之间用逗号隔开。例如：

```
int a[8] = {34, 25, 18, 99, 5, 18, 60, 1};
```

上述初始化过程中，数组元素的次序与括号内各数值的次序严格对应，因此赋值后的结果应为：`a[0] = 34`，`a[1] = 25`，`a[2] = 18`，`a[3] = 99`，`a[4] = 5`，`a[5] = 18`，`a[6] = 60`，`a[7] = 1`。

☐ 数组定义时，可以只给其中的一部分元素赋值。例如：

```
int a[8] = {34, 25, 18};
```

上述语句仅仅对数组中的前 3 个元素进行了赋值，而后面剩余的元素则被默认置为 0。也就是说当对数组中的部分元素初始化时，没有被初始化的另一部分元素就会被置为 0。基于这样的道理可以利用下面的语句来直接将数组中所有元素都初始化为 0：

```
int a[8] = {0};
```

在上述语句中，大括号中仅有 1 个 0，而整个数组中的元素就都被置为 0 了，但是这并不意味着下面这条语句会把数组中所有的元素都初始化为 25：

```
int a[8] = {25};
```

上面这条语句仅仅将数组中的第一个元素置为了 25，而其他元素仍将被默认设为 0。

☐ 在声明数组时若对其中全部元素均赋初值，那么也可以不指定数组的具体长度，编译器会根据赋值的情况为数组自动指定合适的长度。因此下面两条语句其实是等价的：

```
int a[8] = {34, 25, 18, 99, 5, 18, 60, 1};  
int a[] = {34, 25, 18, 99, 5, 18, 60, 1};
```

6.1.4 小心访问越界

在 C 语言中使用数组的一个潜在危险就是 C 并不提供对于数组边界检查。事实上, 如果有一个容量为 10 的数组, 读者可以尝试着去访问第 11 个值, 甚至第 100 个值。这样做有些时候程序并不会崩溃。这取决于第 11 个值或第 100 值的性质。请看下面这个例子:

```
#include <stdio.h>

void main() {

    int a[5] = {1, 2, 3, 4, 5};
    int b[5] = {6, 7, 8, 9, 10};

    printf("%d\n", b[6]);
}
```

图 6.3 是程序的运行结果。

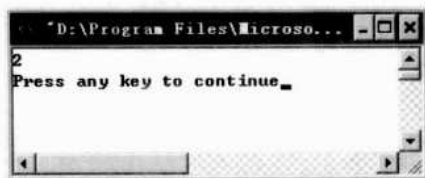


图6.3 数组的越界访问

很明显, 数组 **b** 中不存在 **b[6]** 这一项, 但是由于 **b[6]** 这个位置存在一个合法值, 于是程序仍然输出了结果 (事实上 **b[6]** 的位置存放的是 **a[1]** 这个值)。尽管程序没有崩溃但至少发生了逻辑错误。如果非法位置上的值是比较敏感的, 那么后果可能非常严重。在程序设计时务必杜绝这种情况的发生。

6.1.5 数组应用举例

1. 数据的基本统计分析

在实际的 C 语言开发项目中不使用数组几乎是不可能! 数组提供了一种简单有效的数据组织方式, 它能帮助我们实现许多看起来比较复杂的操作。当我们掌握一组数据之后, 可能就希望对这些数据做进一步的分析。

例如现在班上有 10 名学生, 他们的一次考试成绩如下 (Name(Score)): LiPing(90)、JinHua(85)、ZhangYumei(98)、LiuYue(73)、SongLei(82)、WangWei(75)、WangHui(81)、LiMing(61)、GuoAiping(88)、MengYaolin(86)。

现在我们希望得到的统计结果包括平均分、标准差和每个人距平均成绩的离散情况。平均值表征了这次考试全体学生考试成绩的平均情况。标准差则是测量这些成绩偏离平均值的程度, 这衡量出了这个班级学生之间成绩的差距是否较大。而每个人距平均成绩的离散情况则能够针对不同的个人找出他们的成绩与平均成绩比较相差的具体程度。

标准差 **st_dev** 的计算公式如下:



$$\text{st_dev} = \sqrt{\frac{\text{sum_sqr}}{n+1} - \text{mean}^2}$$

其中

$$\text{sum_sqr} = \text{list}[0]^2 + \text{list}[1]^2 + \cdots + \text{list}[n]^2 = \sum_{i=0}^n \text{list}[i]^2$$

$$\text{mean} = \text{sum} / (n+1)$$

$$\text{sum} = \text{list}[0] + \text{list}[1] + \cdots + \text{list}[n] = \sum_{i=0}^n \text{list}[i]$$

下面我们就编程实现上述数据统计分析功能，代码清单如下：

```
#include <stdio.h>
#include <math.h>

#define SIZE 10

void main()
{
    double list[SIZE], /*待分析的数据表*/
           mean,       /*平均值*/
           sum,        /*总和*/
           st_dev,     /*标准差*/
           sum_sqr;    /*平方和*/

    int i = 0;
    printf("请输入待分析的%d个数据，并以空格隔开\n", SIZE);
    for(; i < SIZE; i++)
        scanf("%lf", &list[i]);

    sum = 0.0;
    sum_sqr = 0.0;
    for(i = 0; i < SIZE; ++i)
    {
        sum += list[i];
        sum_sqr += list[i] * list[i];
    }

    mean = sum / SIZE;
    st_dev = sqrt(sum_sqr / SIZE - mean * mean);

    printf("平均值: %.2f\n", mean);
    printf("标准差: %.2f\n", st_dev);
    printf("\n个人成绩相对于平均值的离散度: \n");
    printf("编号      成绩      离散度\n");
    for(i = 0; i < SIZE; ++i)
        printf("%3d%4c%9.2f%5c%9.2f\n", i, ' ', list[i], ' ', list[i]-mean);
}
```

请读者完成编码后编译并运行上述程序。

2. 用冒泡法对数据排序

冒泡算法是用来对数据进行排序的经典算法之一。它的基本思想是模拟气泡的上浮过程。可以把数组想象成一个竖立的容器，有很多的泡泡（数据元素），大泡泡会首先浮上去。设待排序的数组中有 n 个元素：

第1步 比较待排序元素 $a[n-1]$ 和 $a[n-2]$ 的值，如果 $a[n-1]$ 的值小于 $a[n-2]$ 的值，则交换 $a[n-1]$ 与 $a[n-2]$ 。

第2步 对 $a[n-i]$ 和 $a[n-i-1]$ 进行同样的操作，直到对 $a[0]$ 和 $a[1]$ 做完操作，如图 6.4 所示演示了这个过程。

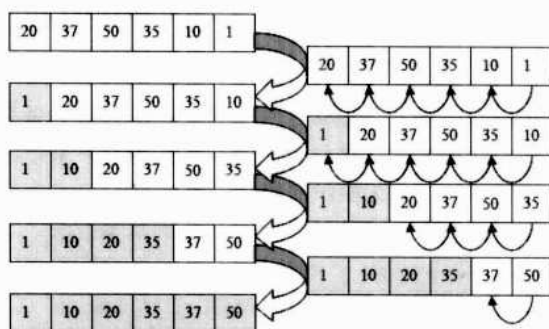


图6.4 冒泡法对数据进行排序

下面就编程实现冒泡排序算法，代码清单如下：

```
#include <stdio.h>
#define SIZE 6

int main()
{
    int a[] = {20, 37, 50, 35, 10, 1};
    int i, j, tmp;
    for(j = 0; j < SIZE; j++)
        for(i = SIZE-1; i > j; i--)
            if(a[i] < a[i-1])
            {
                tmp = a[i];
                a[i] = a[i-1];
                a[i-1] = tmp;
            }

    for(i = 0; i < SIZE; i++)
        printf("%d ", a[i]);

    return 1;
}
```

请读者完成编码后编译并运行上述程序。




6.2 数组类型的参数

上一章中我们介绍了函数的使用，函数中的参数可以是各种类型的数据，那么数组可不可以作为函数的参数来使用呢？这一节我们就来解决这个问题。

6.2.1 以数组作为参数

数组可以作为函数的参数来使用。这时，函数调用的实参列表里出现的数组参数只是一个无下标的数组名。此时，传递给函数的值实际上只是数组元素的起始地址。当数组以不带下标的数组名的形式被传进函数时，函数体内就可以使用带下标的形参来存储数组元素。

 **注意：**出现在形参列表中的数组后面需要跟一个方括号。

函数所操作的是原始数组，而非其副本，这与我们在前一章中所介绍的内容似乎有些出入，但这并非是一种矛盾。因为我们已经强调过数组名作为实参来使用时被传递的实际值就是“地址”，如果传递的是地址，那么操作的就是原值。

下面这段示例代码演示了将数组作为参数进行传递的基本方法：

```
#include <stdio.h>

void output(int array[], int num)
{
    int i = 0;
    printf("The elements in the array are:\n");
    for(; i < num; i++)
        printf("%d ", array[i]);
}

void main()
{
    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    output(a, 10);
}
```

观察上述代码，读者可能注意到了在函数的形参列表中，数组参数的声明形式非常特别，因为方括号内没有指明具体元素的个数。这是因为 C 语言中不会生成实参数组的副本，也就意味着不会开辟新的存储空间，所以这时也就无须指定数组的大小。

另外，函数调用语句：

```
output(a, 10);
```

也可以写作：

```
output(&a[0], 10);
```

两者完全等价。

在函数声明的形参列表中，下面两种数组参数声明方法具有相同的作用：

```
void output(int array[], int num);
void output(int * array, int num);
```


也就是说,“int array[]”和“int * array”具有相同的作用。第一种方式表明实参是一个数组。第二种则更直接地表明了传递的是一个参数。因为 C 语言中通过传递数组的首地址来完成一个数组参数的传递,因此这两种方式等价。

6.2.2 避免数组被修改

将数组用作参数的时候,由于传递的是数组的首地址,因此在函数体中操作的是数组本身,而非其副本,这样数组元素就有可能被修改。但是有些时候我们不希望数组在其他函数体中被修改。在 ANSI C 中提供了一个支持这种基本保护的语法,下面是两个以数组为参数的函数声明语法形式:

```
返回值类型 函数名(const 元素类型 数组名[]);  
返回值类型 函数名(const 元素类型 * 数组名);
```

即在函数的形参数组声明中包含关键词 `const`, 该种语法用于通知 C 编译器该数组只是函数的一个输入, 因此该数组不应当在函数中被修改。例如下面这段示例代码中, 函数 `process()` 使用一个数组作为参数, 且该参数被关键词 `const` 所修饰。由于函数 `process()` 中试图修改数组内容, 因此编译器将抛出编译错误。

```
#include <stdio.h>  
  
int process(const int array[])  
{  
    array[2] = array[3] + array[4];  
    return array[2];  
}  
  
int main()  
{  
    int a[] = {23, 41, 0, -99, 2};  
    int result = process(a);  
    printf("%d\n", result);  
    return 1;  
}
```

请读者完成编码后编译该程序, 在 Visual C++ 中, 编译器给出了如下错误提示: “error C2166: l-value specifies const object”。可见, 该关键词使编译器在函数中任何试图修改数组元素的地方上报语法错误, 如图 6.5 所示。

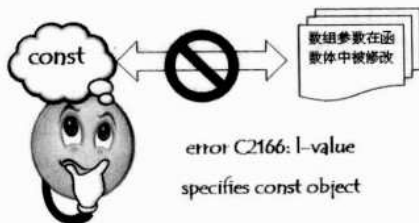


图6.5 关键字const保护数组参数不被修改

在函数的形参列表中, 关键字 `const` 指出所声明的数组变量是一个严格的输入参数, 并



且不能在函数体中修改。我们需要再次强调，在函数声明中所引用的形参数值将会是实参数组的地址，所以如果省略了关键字 `const`，那么当函数体中试图修改该参数值时，原数组就会随之改变。因此使用关键词 `const` 来限制数组不被其他函数所修改是一种非常好的编程风格，它可以使程序更加健壮，能够有效地防止一些意外修改的发生。

S.3

6.2.3 函数返回数组的两种方法

在介绍函数的时候，我们就已经强调过：函数可以有多个参数，但是返回值只能有 1 个，如图 6.6 所示。鉴于此就可以推断出函数的返回值绝对不可以是数组，因为数组是多个数据形成的集合。但是有时候我们就想让函数返回多个返回值或者返回一个数组，那有没有什么解决办法呢？

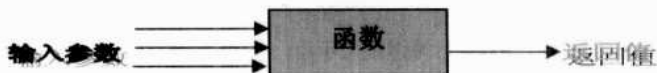


图6.6 函数返回值模型

1. 使用一个输出参数将结果数组返回至调用模块

由于调用模块中的数组能够在函数中被修改，借助这个特性，我们在调用模块中定义的数组以参数的形式传到函数中，函数再将操作的结果存储在这个数组中。当函数返回后，数组中的值并不会丢失，于是我们也就实现了返回数组的目的。

下面来举例说明这种方式。假设有一组数，共有 `SIZE` 个，且数字之间大小均不相同。现在希望求出这组数中最大的 `NUM` 个数。在下面的代码中，函数 `getBigger()` 用来完成这个功能。为了设计实现这个函数，需要考虑到问题有以下几个：

- ❑ 因为题目要求求得数组中最大的 `NUM` 个数，因此返回值有可能不止 1 个，于是想到让函数返回一个数组，数组的容量为 `NUM`。于是在主函数中我们声明了一个数组 `result`，并将其作为参数传递给函数 `getBigger()`。函数体内会将得到的结果存进数组 `result` 中，当函数返回后，数组 `result` 中的值就是我们所求的结果。
- ❑ 要设计一个能够实现该功能的算法。最容易想到的是把原数组按从大到小的原则进行排序，然后把排序后数组中的前 `NUM` 个数赋给数组 `result`。但是这样一来原数组的状态就会被改变，这是我们不希望看到的，为了防止原数组被意外改变，我们在形参声明列表中使用了关键字 `const`。
- ❑ 我们要来设计真正符合自己要求的算法。这里所采用的算法具体是这样的，首先从数组中挑出最大的那个值，再将其赋给一个临时变量 `LastValue`，然后继续求解剩余的 `NUM-1` 个目标值。在求解每个剩余目标值时，我们都要求当前这个值是除了上一个求得值以外最大的，于是我们就需要使用临时变量 `LastValue` 来存储上一个求得的数值，并不断更新它，当然，该变量的初值应该是数组中最大的那个数。

以下是完整的代码清单：

```
#include <stdio.h>

#define SIZE 8
#define NUM 3
```

```
bool getBigger(const int array[], int result[])
{
    if (NUM>SIZE)
        return false;

    int j, k, LastValue;
    j = 0;
    result[j] = array[0];
    //计算序列中数值最大的项
    for(k = 0; k < SIZE; k++)
        if(result[j] < array[k])
            result[j] = array[k];
    LastValue = result[j];

    //计算序列中前面几个较大的项
    for(j = 1; j < NUM; j++)
    {
        for(k = 0; k < SIZE; k++)
            if(result[j] < array[k] && array[k] < LastValue)
                result[j] = array[k];
        LastValue = result[j];
    }

    return true;
}

int main()
{
    int a[SIZE] = {23, 41, 0, -99, 2, 51, 39, 3};
    int result[NUM];
    if(getBigger(a, result)==true)
    {
        int i = 0;
        for(; i < NUM; i++)
            printf("%d\n", result[i]);
    }
    else
        printf("Error! Please check.\n");

    return 1;
}
```

2. 让函数返回一个指针

下面的大整数乘法实现程序非常典型，它不但使用数组作为参数进行传递，而且还让函数返回了数组。

由于计算机的精度有限，因此单纯使用程序设计语言提供的原子数据类型来完成两个大整数的乘法显然不切实际，所以考虑用两个数组分别存储一些小于 10 的整数，这些数字按顺序排列在一起，分别表示一个大整数的每一位上的数字，然后按照基本乘法规则对这两个



大整数进行运算。下面给出了程序的代码清单：

```
#include <stdio.h>
#include <memory>

// 返回位数为 size1+size2
int* multi(int* num1, int size1, int* num2, int size2)
{
    int size = size1 + size2;
    int* ret = (int *)malloc(size*sizeof(int));
    int i = 0;

    memset(ret, 0, sizeof(int) * size);

    for (i = 0; i < size2; ++i)
    {
        int k = i;
        for (int j = 0; j < size1; ++j)
        {
            ret[k++] += num2[i] * num1[j];
        }
    }

    for (i = 0; i < size; ++i)
    {
        if (ret[i] >= 10)
        {
            ret[i+1] += ret[i] / 10;
            ret[i] %= 10;
        }
    }

    return ret;
}

int main()
{
    int num1[] = {1,2,3,4,5,6,7,8,9,1,1,1,1,1}; //第一个大整数 11111987654321
    int num2[] = {1,1,1,2,2,2,3,3,3,4,4,4,5,5}; //第二个大整数 55444333222111

    int* ret = multi(num1, 14, num2, 14);

    for (int i = 27; i >= 0; i--)
    {
        printf("%d", ret[i]);
    }

    free(ret); //释放内存

    return 0;
}
```

完成编码后，编译并运行程序即可。

6.3 多维数组的使用

数组中的元素可以是任意类型的，当然也可是数组类型的。通过这种数组的嵌套可以将一维数组向上扩展成为多维数组。二维数组、三维数组……都是可以的。二维数组是应用最多的多维数组形式，本节将主要讲解二维数组的使用方法，其他多维数组可以以此为基础推得，所以本节将不会过多地研究它们。

6.3.1 从一维到二维

在欧几里德几何中，线是一维的，面是二维的，体是三维的。数轴上的一个点仅仅由一个坐标就可以唯一确定，而二维平面上的点则需要由横坐标和纵坐标这一对坐标来确定。于是我们想到可不可以把一维数组也上升到二维数组，甚至更高的维度。C语言中允许多维数组的存在。

二维数组是一种常用的多维数组。二维数组相当于平面内的一个二维矩阵。二维数组定义的一般形式如下：

类型说明符 数组名[常量表达式 1][常量表达式 2];

例如：

```
float array[2][3]; //定义一个2×3的浮点型数组
```

array[2][3]表示的是2行3列，即第一个参数表示行数，第二个参数表示列数。同样可以通过下标来对数组中的任意元素进行操作。但是因为数组的下标仍然是从0开始的，所以数组array中不存在array[2][3]、array[2][1]和array[0][3]等元素。

二维数组也可以看作是一种特殊的一维数组，只不过这个特殊的一维数组的元素仍然是一个数组，或者说二维数组就是一维数组再嵌套一维数组形成的。例如对于前面定义的数组array，我们就可以认为该数组具有两个元素，即array[0]和array[1]，而这两个元素每个又都是包含了3个元素的一维数组。于是二维数组array[2][3]就可以被看做是两个以数组为元素的复合数组，如图6.7所示。

在C语言里，二维数组中元素排列的顺序是后面一行的行首元素在内存中的位置将紧挨着前一行中的行尾元素，换句话说，二维数组是按行依次存储的。图6.8所示为array[2][3]数组在内存中的存放顺序。

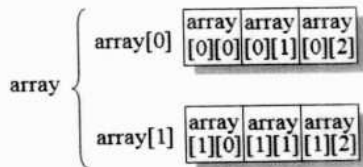


图6.7 对于二维数组的理解

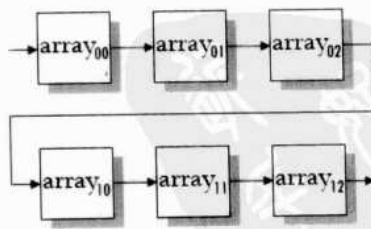


图6.8 二维数组存放顺序

6.3.2 初始化及使用二维数组

首先我们来看一下如何初始化一个二维数组。可是采用的方法共有4种。



- ☑ 直接将二维数组中的所有元素按照其实际排列顺序进行赋值, 这时也就是把二维数组简单地看成是一个一维数组 (是将二维数组的每行行首接到上一行的行尾所形成的一维数组)。例如:

```
int array[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

- ☑ 上面的方法可读性很差, 因为数组之间的行列关系非常模糊, 所以我们推荐使用下面这种方法: 将二维数组的每行分别用大括号括起来, 然后分行给二维数组赋值。例如:

```
int array[3][4] = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}};
```

- ☑ 可以选择性地对部分数组元素进行赋值, 特别是对于数组容量比较大的情况。例如:

```
int array[2][3] = {{1}, {4}};
```

上面的语句的作用只对每行首列进行了赋值, 其他元素被默认赋值为 0, 它对应的矩阵如下:

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 0 & 0 \end{bmatrix}$$

当然也可以选择某行中的任意一列或多列来进行赋值, 例如:

```
int array[2][3] = {{1}, {0, 4}};
```

它对应的矩阵如下:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \end{bmatrix}$$

用这种方法对稀疏矩阵进行赋值特别方便。也可以跳过其中的某行, 对其他行进行赋值, 例如:

```
int array[2][3] = {{}, {0, 4}};
```

它对应的矩阵如下:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 4 & 0 \end{bmatrix}$$

- ☑ 如果在初始化时对二维数组里的全部元素进行了赋值, 则可以省略第一维的长度, 但第二维的长度不能省略。例如:

```
int array[][3] = {1, 2, 3, 4, 5, 6};
```

为了进一步说明二维数组的使用方法, 下面举一个简单的例子, 该例子实现的功能是对一个 4×4 的二维数组进行转置:

```
#include <stdio.h>

void main() {

    int a[4][4] = {{0, 1, 2, 3}, {4, 5, 6, 7},
                  {8, 9, 10, 11}, {12, 13, 14, 15}};

    int i = 0;
    int j = 0;
    int tmp = 0;

    for(i = 0; i < 4; i++)
```



```
{
    for(; j < 4; j++)
    {
        tmp = a[i][j];
        a[i][j] = a[j][i];
        a[j][i] = tmp;
    }
    j=i+1;
}

for(i = 0; i < 4; i++)
{
    for(j = 0; j < 4; j++)
    {
        printf("%d ", a[i][j]);
    }
    printf("\n");
}
```

通过二维数组的学习，可以很容易推广到高维数组的情况，多维数组的通用定义如下：

类型说明符 数组名[常量表达式 1][常量表达式 2]…；

有兴趣的读者可以自己编写一些简单的程序来试用一下高维数组，这里就不再赘述了。

6.3.3 多维数组应用举例

计算机中的位图有时也称为位图图像或点阵图像，它由像素组成，可以看作是一种二维的栅格，每个栅格中被填充不同的颜色，当这种栅格的尺寸足够小的时候，肉眼便无法分辨出具体的一个栅格，这样我们所观察到的就是一副过度平缓自然的数字图像。每个有颜色的栅格就是一个像素。

位图是 Windows 显示图像的基础。当位图被放大到一定倍数时，就可以看见赖以构成整个图像的方格，这时平滑的线条和形状会显得参差不齐。如图 6.9 所示的是一幅典型的位图，其中右图为较小尺寸的位图，左图为放大后的效果，从中可以很容易地看到位图由许多被填充了一定颜色的方格组成。

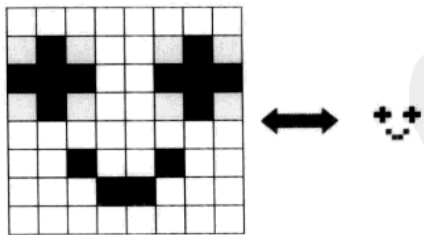


图6.9 位图

当我们理解了位图这种二维栅格的本质之后，就可以用一个二维数组来存放位图信息。二维数组所形成的矩形就是图像的全貌，数组中的每个元素都表示特定行列上像素点的颜色。以这个二维数组所形成的矩阵为基础，可以实现对图像进行变换处理，本节就以此来介



绍多维数组的应用。

1. 图像的水平镜像效果

图像的镜像变换是数字图像几何变换中最简单的一种变换方式，之所以称为镜像变换，是因为图像经过处理后的效果就如同照镜子一样。镜像变换分为两种，即水平镜像和垂直镜像。

水平镜像变换就是以图像垂直中线为轴，将图像的所有像素进行对称变换。换句话说，就是将图像的左半部和右半部进行对调，其效果如图 6.10 所示。可见，水平镜像变换产生原始图像的水平投影，就像图像在镜子中的显示效果一样。

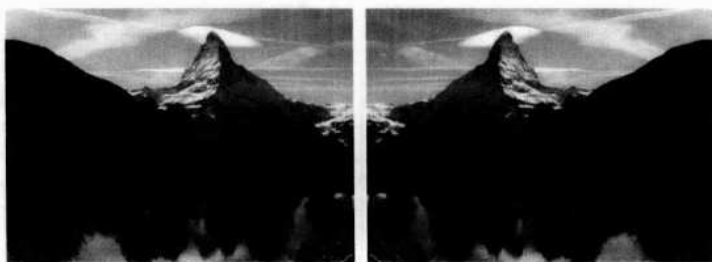


图6.10 图像的水平镜像效果

为了便于理解，这里假设图像的像素矩阵如下，其中用 $N_x N_y$ 标记该像素在原始图像中的位置，即第 N_x 行和第 N_y 列。

```
00 01 02 03 04 05
10 11 12 13 14 15
20 21 22 23 24 25
30 31 32 33 34 35
40 41 42 43 44 45
50 51 52 53 54 55
```

经过水平镜像变换后的图像像素矩阵如下：

```
05 04 03 02 01 00
15 14 13 12 11 10
25 24 23 22 21 20
35 34 33 32 31 30
45 44 43 42 41 40
55 54 53 52 51 50
```

下面我们就编程实现图像的水平镜像处理功能，其代码清单如下，代码中使用一个二维数组来表示位图矩阵：

```
#include <stdio.h>

#define WIDTH 6
#define LENGTH 6

int main()
```

```
{
    int pic[LENGTH][WIDTH] = {{0, 1, 2, 3, 4, 5},
                                {10, 11, 12, 13, 14, 15},
                                {20, 22, 22, 23, 24, 25},
                                {30, 31, 32, 33, 34, 35},
                                {40, 41, 42, 43, 44, 45},
                                {50, 51, 52, 53, 54, 55}};

    int i, j, tmp;
    for(i = 0; i < LENGTH; i++)
        for(j = 0; j < WIDTH/2; j++)
        {
            tmp = pic[i][j];
            pic[i][j] = pic[i][WIDTH-1-j];
            pic[i][WIDTH-1-j] = tmp;
        }

    printf("水平镜像处理后的结果如下: \n");
    for(i = 0; i < LENGTH; i++)
    {
        for(j = 0; j < WIDTH; j++)
            printf("%d ", pic[i][j]);
        printf("\n");
    }

    return 1;
}
```

请读者完成编码后编译并运行上述程序。

2. 图像的垂直镜像效果

垂直镜像变换同样也是对图像进行像素的对称变换,但不同的是垂直镜像变换不再以垂直中线为轴而是以图像的水平中线为轴进行上半部分和下半部分的对调,效果如图6.11所示。可见,垂直镜像变换让图像在垂直方向上进行投影,效果好比在水中的影子。



图6.11 图像的垂直镜像效果

为了便于理解,这里假设图像的像素矩阵如下,其中用 $N_x N_y$ 标记该像素在原始图像中的位置,即第 N_x 行和第 N_y 列。



```
00 01 02 03 04 05
10 11 12 13 14 15
20 21 22 23 24 25
30 31 32 33 34 35
40 41 42 43 44 45
50 51 52 53 54 55
```

经过垂直镜像变换后的图像像素矩阵如下:

```
50 51 52 53 54 55
40 41 42 43 44 45
30 31 32 33 34 35
20 21 22 23 24 25
10 11 12 13 14 15
00 01 02 03 04 05
```

垂直镜像处理功能的代码清单如下:

```
#include <stdio.h>

#define WIDTH 6
#define LENGTH 6

int main()
{
    int pic[LENGTH][WIDTH] = {{0, 1, 2, 3, 4, 5},
                                {10, 11, 12, 13, 14, 15},
                                {20, 22, 22, 23, 24, 25},
                                {30, 31, 32, 33, 34, 35},
                                {40, 41, 42, 43, 44, 45},
                                {50, 51, 52, 53, 54, 55}};

    int i, j, tmp;
    for(i = 0; i < WIDTH; i++)
        for(j = 0; j < LENGTH/2; j++)
        {
            tmp = pic[j][i];
            pic[j][i] = pic[LENGTH-j-1][i];
            pic[LENGTH-j-1][i] = tmp;
        }

    printf("垂直镜像处理后的结果如下: \n");
    for(i = 0; i < LENGTH; i++)
    {
        for(j = 0; j < WIDTH; j++)
            printf("%d ", pic[i][j]);
        printf("\n");
    }
}
```

```
    return 1;
}
```

请读者完成编码后编译并运行上述程序。

6.4 字符数组

数组里面存放的数据可以是任意类型的，当然也可以存储字符。用来存储字符的数组称为字符数组。现实中的文本数据可能是海量的！因为文字是最直接、最简单、最准确的信息记录方式。报刊中的文章、个人简历中的信息以及计算机中用来指导用户操作的提示语都是文本信息。可想而知，字符数组的作用是多么重要。

6.4.1 定义与初始化

字符数组只是数组的一种特殊类型，因此数组所具有的普遍属性，字符数组也同时具有。定义和初始化字符数组的方式和处理普通数组的方式相同，例如下面这段代码：

```
char c[10];
c[0] = 'G'; c[1] = 'o'; c[2] = 'o'; c[3] = 'd'; c[4] = ' ';
c[5] = 'l'; c[6] = 'u'; c[7] = 'c'; c[8] = 'k'; c[9] = '!';
```

上述代码等价于：

```
char c[10] = {'G', 'o', 'o', 'd', ' ', 'l', 'u', 'c', 'k', '!'};
```

上述代码还可以写成：

```
char c[] = {'G', 'o', 'o', 'd', ' ', 'l', 'u', 'c', 'k', '!'};
```

上述语句所得到的数组示意如图 6.12 所示。

G	o	o	d	空格	l	u	c	k	!
---	---	---	---	----	---	---	---	---	---

图 6.12 数组示意1

需要说明的是，在初始化数组时，如果大括号中的字符个数大于数组的长度，那么就会产生语法错误！如果初始的字符个数小于数组的长度，那么就按顺序先对数组中前面的元素进行赋值，其余的元素则自动被设置为空字符，即'\0'。例如下面这段代码：

```
char c[10] = {'H', 'e', 'l', 'l', 'o', '!'};
```

上述代码所得到的数组示意如图 6.13 所示。

H	e	l	l	o	!	\0	\0	\0	\0
---	---	---	---	---	---	----	----	----	----

图 6.13 数组示意2

另外，二维及多维字符数组也是允许的，这点与普通数组相同，这里就不再赘述了。

6.4.2 字符串的使用

1. C 语言中的字符串

在 C++ 或者 Java 这样的面向对象语言中一般会提供一个封装比较完好的字符串类型，



但 C 语言中并没有这样做。C 语言中的字符串就是以一维字符数组的形式来实现的。例如我们可以在一个字符数组中存放一个字符串“Good Luck!”, 那么这个数组中将含有 10 个字符, 这个长度就是该字符串的长度。

数组的长度和字符串的长度有时无法等同看待。例如在图 6.13 中的一个字符数组, 其长度为 10, 但是实际字符串仅仅占用了该字符数组的 6 个位置, 也就是说这时字符串的长度和字符数组的长度并不相等。

另外, 我们更应当关心的是字符串的实际长度而非字符数组的长度。例如, 当我们希望编写一个用以记录来访者留言的留言本程序时, 因为不知道每个来访者会写多长的一句话, 所以我们可以为每个留言者设置一个长度为 1000 的字符数组用来存放不同的人的留言。

因为, 每个留言者的留言长度都是不同的, 因此当客户需要查看历史留言的时候, 我们会使用一些输出语言来输出不同的留言。这时我们并不会把每个长度为 1000 的字符数组中的全部字符都输出, 因为这个数组里面可能存在很多空字符。

这时我们更关心的就是字符数组中存放字符串的实际长度。为了满足实际中这种要求, C 语言规定了一个“字符串”结束标识, 即字符‘\0’。一旦系统察觉到这个标志则认为字符串已经结束。例如, 当一个字符数组的 100 个字符元素是‘\0’, 那么这就意味着该字符串的实际长度仅为 99。

2. “字符串”结束标识——字符‘\0’

在 ASCII 码表中, 字符‘\0’的码值为 0, 它不是一个可以显示的字符, 更准确地说是一个“空字符”, 或者“空操作”, 因此选择它来作为字符串的结束标识是非常适合的。而且系统还会为字符串常量自动追加该字符, 例如有一个字符串常量“Good Luck!”, 该串中共有有效字符 10 个, 但在内存中它实际占用 11 个字节, 因为最后一个字符‘\0’是系统自动追加的。在实际编程中, 字符‘\0’往往被用来判断字符串是否结束, 这是一个很常用的编程方法。

3. 字符串初始化的其他方法

我们已经学习了有关字符串初始化的基本方法, 但是 C 语言中还提供了一些其他的方法, 例如可以将整个字符串用双引号括起来再放到大括号中:

```
char str[] = {"Good Luck!"};
```

大括号也可以被省略, 于是上面的初始化也可以写成:

```
char str[] = "Good Luck!";
```

这种方法将整个字符串用作一个初始值, 而非是对单个的字符进行逐个赋值, 这样既简便又符合人们理解习惯, 这是一种值得推荐的语法方式。但是需要读者注意的是, 当使用一个字符串作为初始值时, 系统会自动在字符串后面添加结束符‘\0’。因此上述字符串初始化语句应该和下面这条语句等价:

```
char str[] = {'G', 'o', 'o', 'd', ' ', 'L', 'u', 'c', 'k', '!', '\0'};
```

但是前面的语句和下面这条语句则不同, 这点读者务必留意:

```
char str[] = {'G', 'o', 'o', 'd', ' ', 'L', 'u', 'c', 'k', '!'};
```

如果在数组声明时规定了其长度，那么一旦字符串的实际长度小于定义的长度，系统就会把剩余的空位都用字符'\0'来填充，例如下面这行代码：

```
char str[12] = {"Good Luck! "};
```

该代码生成的实际字符数组情况如图 6.14 所示。

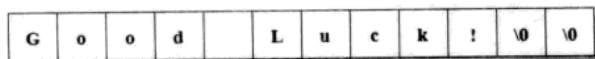


图6.14 字符数组的实际内容

另外需要说明的是，字符数组初始化时末尾字符是否是结束符'\0'都可以，具体要不要追加'\0'完全取决于实际情况，因此下面的初始化语句是完全合法的：

```
char str[10] = {'G', 'o', 'o', 'd', ' ', 'L', 'u', 'c', 'k', '!'};
```

6.4.3 字符串的处理——大小写转换函数

在 C 语言中，可以使用 `strlwr()` 函数将字符串转换为小写形式，该函数的原型如下：

```
extern char *strlwr(char *s);
```

该函数只转换参数 `s` 中出现的大写字母，不改变其他字符。返回值指向字符串 `s` 的指针。与 `strlwr()` 函数相对应，C 语言中还提供了函数 `strupr()` 将字符串转换为大写形式，该函数原型如下：

```
extern char *strupr(char *s);
```

同样，只转换 `s` 中出现的小写字母，不改变其他字符，并返回指向 `s` 的指针。

6.4.4 字符串的处理——字符串比较函数

字符串比较是在应用字符串解决问题时常用的方法之一。在 C 语言中通过函数 `strcmp()` 实现字符串的比较功能，该函数的原型如下：

```
extern int strcmp(char *s1, char *s2);
```

该函数对字符串 `s1` 和 `s2` 进行比较。如果 `s1 < s2` 时，则返回值小于 0；如果 `s1 = s2` 时，则返回值等于 0；如果 `s1 > s2` 时，则返回值大于 0。

如何定义 `s1` 和 `s2` 哪个大，哪个小呢？字符串的比较规则是这样的：如图 6.15 所示的是对两个字符串从左到右逐个字符进行 ASCII 值的比较，直到出现不同的字符或者遇到 '\0' 为止。如果两个字符串的全部字符都相等，那么这两个字符串就相等。如果在比较过程中出现了不同的字符，那么第一个不同字符的 ASCII 值比较结果就是这两个字符串的比较结果。

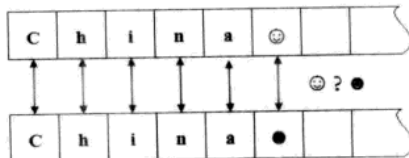


图6.15 字符串的比较

例如下面这段示例程序演示了函数 `strcmp()` 的使用方法：



```
#include <stdio.h>
#include <string.h>

int main()
{
    char s1[] = "Comes the moment, comes the man!";
    char s2[] = "Comes the moment, comes the man!";

    int result = strcmp(s1,s2);

    if(!result)
        printf("s1 and s2 are identical.\n");
    else if(result<0)
        printf("s1 is less than s2.\n");
    else
        printf("s1 is bigger than s2.\n");

    return 0;
}
```

另外还需要提醒读者注意：对两个字符串进行比较时只能使用函数 `strcmp()`，而不能使用 “==”，因此下面这段代码是错误的：

```
if(s1 == s2) {...}
```

正确的形式应该写为：

```
if((strcmp(s1, s2))==0) {...}
```

6.4.5 字符串的处理——字符串长度的获得

函数 `strlen()` 是用来测试字符串长度的函数，该函数的原型如下：

```
extern int strlen(char *s);
```

该函数的返回值是字符串的实际长度（注：不包含 '\0' 在内）。例如下面这段示例代码：

```
#include <stdio.h>
#include <string.h>

int main()
{
    char s[25] = "My heart will go on!";
    printf("%s\n" has %d chars.\n",s,strlen(s));

    return 0;
}
```

运行上述程序，程序输入结果为 “My heart will go on!” has 20 chars.”。

6.4.6 字符串的处理——字符串连接函数

函数 `strcat()` 的作用是把两个字符数组中的字符串连接起来，其函数的原型如下：

```
extern char *strcat(char *dest, char *src);
```


函数把指针 `src` 所指字符串添加到 `dest` 结尾处, 覆盖 `dest` 结尾处的 `'\0'`, 并在整个字符串的末尾添加 `'\0'`。需要说明的是: `src` 和 `dest` 所指内存区域不可以重叠且 `dest` 必须有足够的空间来容纳 `src` 的字符串。该函数最后返回一个指向 `dest` 的指针。例如下面这段代码:

```
char dest[12] = "Good ";
char src[] = "Luck!";
strcat(dest, src);
```

执行上述操作后, 字符串 `dest` 中的内容如图 6.16 所示。

G	o	o	d		L	u	c	k	!	\0	\0
---	---	---	---	--	---	---	---	---	---	----	----

图6.16 字符串`dest`中的内容

再次提醒读者注意, 字符数组 `dest` 必须有足够的空间来容纳 `src` 的字符串, 在上例中 `dest` 的长度为 12, 所以长度足够, 如果将其定义修改为:

```
dest[] = "Good";
```

那么程序执行就会出错, 这在使用函数 `strcat()` 时必须注意的一个问题。

6.4.7 字符串的处理——字符串复制函数

C 语言库函数中提供的字符串复制函数主要有两个。

1. 函数 `strcpy()`

`strcpy()` 函数的原型如下:

```
extern char *strcpy(char *dest, char *src);
```

其作用是把指针 `src` 所指的由 `'\0'` 结束的字符串复制到 `dest` 所指的字符数组中。该函数返回一个指向 `dest` 的指针。例如下面这段代码:

```
char dest[10], src[] = "Hello!";
strcpy(dest, src);
```

执行上述操作后, 字符串 `dest` 中的内容如图 6.17 所示。

H	e	l	l	o	!	\0	\0	\0	\0
---	---	---	---	---	---	----	----	----	----

图6.17 字符串`dest`中的内容

在使用 `strcpy()` 函数时, 有以下几点需要说明:

- ☑ 字符数组 `src` 和 `dest` 所指内存区域不可以重叠且 `dest` 必须有足够的空间来容纳 `src` 的字符串, 也就是说字符数组 `dest` 的长度不应该小于字符数组 `src` 的长度。
- ☑ 字符数组 `src` 必须是以数字名的形式出现, 但是字符串 `dest` 可以是字符数组名, 也可以是一个字符串常量。例如下面这条语句也是符合语法规范的:

```
strcpy(dest, "Hello!");
```



- ❑ 赋值运算符不能用于将一个字符串常量或者字符数组赋给另外一个字符数组，在进行字符串复制时只能使用库函数，赋值运算符只能对字符数组中的单个字符变量进行赋值，所以下面两条语句都是错误的：

```
dest = {"Hello!"};  
dest = src;
```

2. 函数 strncpy()

C 语言库函数中提供的另外一个实现字符串复制功能的函数是 `strncpy()`，该函数的原型如下：

```
char * strncpy(char * dest, char *src, unsigned int n);
```

与 `strcpy()` 函数不同，函数 `strncpy()` 可以用来复制字符串的一部分，更准确地说是将字符串 `src` 中最多 `n` 个字符复制到字符数组 `dest` 中，然后返回指向 `dest` 的指针。

`strncpy()` 函数共有 3 个参数：

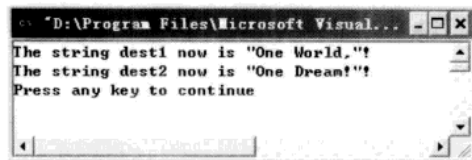
- ❑ 第一个参数是目标字符串。
- ❑ 第二个参数是源字符串。
- ❑ 第三个参数是一个整数。

代表要从源字符串复制到目标字符串中的字符数。另外需要说明的是在使用函数 `strncpy()` 时：如果目标字符数组长度 \geq 参数 `n` > 源字符串长度，那么全部源字符串就会被复制到目标字符串中（包括 `\0`）；如果参数 `n` < 源字符串长度，则在源字符串中按指定长度 `n` 截取复制到目标字符串（不包括 `\0`）；如果参数 `n` > 目标字符数组长度，则发生语法错误。

下面给出一段演示使用函数 `strncpy` 的示例程序：

```
#include <stdio.h>  
#include <string.h>  
  
void main ()  
{  
    char * src = "One World, One Dream!";  
    char dest1[40] = {0}, dest2[40] = {0};  
  
    strncpy(dest1, src, 10);  
    printf("The string dest1 now is \"%s\"!\n", dest1);  
  
    strncpy(dest2, src + (strlen(src)-10), 10);  
    printf("The string dest2 now is \"%s\"!\n", dest2);  
}
```


编译并运行上述程序，输出结果如图 6.18 所示。



6.18 程序运行结果

上面这段示例程序,调用 `strncpy()` 函数时,将源字符串的头 10 个字符复制到 `dest1` 中,这是该函数的最简单且最直接的用法。而第二次使用 `strncpy()` 函数时,程序则将源字符串的后 10 个字符复制到 `dest2` 中,这是一个很有意思的过程。

`strlen()` 函数计算出了 `src` 字符串的长度,即 “`strlen(src)`”。再将 `src` 的长度减去 10 (也就是将要复制的字符数),得出的是截取后字符串 `src` 中剩余的字符数。将 `strlen(src)-10` 和 `src` 的地址相加,即得出指向 `src` 中倒数第 10 个字符的地址的指针,即 `src+(strlen(src)-10)`,并把这个指针作为函数的第二个参数。

 **注意:** 在将字符串 `src` 复制到 `dest1` 和 `dest2` 之前, `dest1` 和 `dest2` 都要被初始化为 ‘\0’。这是因为 `strncpy()` 函数在复制字符串时不会自动将 ‘\0’ 添加到目标字符串后面,因此程序员必须要确保在目标字符串的后面加上 ‘\0’,否则就可能会打印出一些没有意义的乱码。

6.4.8 字符串应用举例

本节将从两个问题入手来帮助读者掌握字符串操作的基本方法。第一个例子是跟实际比较贴近的问题——统计文章中特定单词的出现次数。第二个例子是比较复杂的 ACM 试题。

1. 单词出现次数的统计

通常文字处理软件(例如 MS Office Word 或者 WPS 等)中都提供了查找匹配功能和字数统计功能。下面我们要完成的任务是编写一个程序,实现对一段文本中特定的某个单词进行统计的功能。

这样一个程序首先需要具备字符串的匹配查找功能,然后结合字数统计功能即可达到设计要求。实现此功能的算法也不唯一,在此我们将利用 C 语言库函数中有关字符串操作的一些函数来辅助实现这个功能。这个例子非常典型,因为它反映了字符串操作的一些特有手段和基本方法。下面给出该程序的示例代码清单:

```
#include <stdio.h>
#include <string.h>

int main()
{
    int sum = 0;
    char word[20];
    char text[] = "Now the trumpet summons us again - not \
as a call to bear arms, though arms we need- not \
as a call to battle, though embattled we are-but \
a call to bear the burden of a long twilight struggle, \
year in and year out, rejoicing in hope, \
patient in tribulation - a struggle against \
the common enemies of man: \
tyranny, poverty, disease, and war itself. ";

    char temp[20] = {0};

    printf("Please input the word which you want to count.\n>");
```



```
    gets(word);

    int length = strlen(word);
    int i = 0;
    while(text[i]!='\0')
    {
        strncpy(temp, text+i, length);
        if(strcmp(word, temp)==0)
        {
            sum++;
            i+=length;
        }
        i++;
    }

    printf("The word \"%s\" appears %d times in the text.\n", word, sum);

    return 1;
}
```

完成编码后，编译并运行程序，结果如图 6.19 所示。

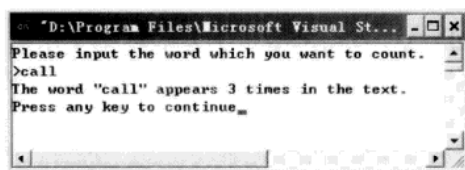


图6.19 程序运行结果

2. 求字符串指数的问题

有两个字符串 a 和 b ，我们定义 $a*b$ 为两个字符串的串联运算。例如，如果 $a="abc"$ ， $b="def"$ ，那么 $a*b="abcdef"$ 。这里可以把串联看成乘法，并将对一个非负整数求幂用一种通常的方式来定义：即 $a^0=""$ （空串），以及 $a^{n+1}=a*(a^n)$ 。

输入：每一次测试例子就是一行可打印的字母序列 S 。 S 的长度在 $1 \sim 1\,000\,000$ 之间。下面给出一段输入示例。

```
abcd
aaaa
ababab
```

输出：对每一个 S 你应该打印出满足 $S=a^n$ 中最大的 n 值。例如对于上面的输入应当可以得到下面的输出：

```
1
4
3
```

这道算法设计问题也是一道 ACM 试题，该题目意思简单明了，显然属于一道字符串匹配问题。尽管题意非常简单，但要求程序在尽可能短的时间内完成计算也是不容易实现的。



这道题目的解答中使用了本章前面所介绍的一些字符串处理函数，具有一定的实际意义。但是由于算法本身比较复杂，考虑到本书读者的不同层次，这里就不做过多的解释了。下面给出程序实现的源代码，供有兴趣的读者参考。

```
#include <stdio.h>
#include <string.h>

char s[2000002];

void main(){
    int i,j,m,n;
    while (gets(s) && strcmp(s, ".") != 0) {
        m = n = strlen(s);
        for (i=2; i<=n; i++) {
            while (n%i == 0) {
                n /= i;
                for (j=0; j<m-m/i && s[j] == s[j+m/i]; j++);
                if (j == m-m/i) m /= i;
            }
        }
        printf("%d\n", strlen(s)/m);
    }
}
```

完成编码后，编译并运行程序即可。





第7章

指 针



C 语言作为一种高级语言，它所编写的 C 程序至今仍能 and 汇编语言程序相媲美，其中一个重要原因就在于 C 语言拥有一种特殊的数据类型——指针。指针既是一种数据类型，也是一种内存操作手段，还可理解成是一种异常强大的语法工具。总之，指针是 C 语言的精髓所在。

读者可能有点不耐烦了，我们不断地给指针冠以各种响亮的名称，却直到本书进行过半之后，才将其抛出，似乎有些“千呼万唤始出来”的意思。这是因为对于很多初学者来说，指针无疑是一个令人望而生畏的概念，指针就是初学者的“噩梦”。

但无论如何学习 C 语言都不可能绕开指针，如果没有指针，那么 C 语言也就不能再称为 C 语言了。何况，对于 C 程序设计高手来说指针还是一把不可或缺、爱不释手的利器。本章就来向读者介绍有关指针的概念及其用法方面的知识。



7.1 指针与地址

指针令很多初学者感到头疼，因为在初学者看来指针是一个虚无缥缈的概念，至少不像整数、字符那样容易理解。理解指针的本质是正确使用它的关键，因此本节的内容对于初学指针的人来说将是非常重要的。

7.1.1 内存和地址的概念

从物理上讲内存就是一块半导体存储材料，如图 7.1 所示。任何需要被计算机执行的程序，包括指令和数据，都要先从外部存储器（包括硬盘、光盘等）调入内存后才能被执行。内存中同时持有很多数据，当前需要哪个数据，CPU 都会从内存中精确地找到，然后再参与运算。

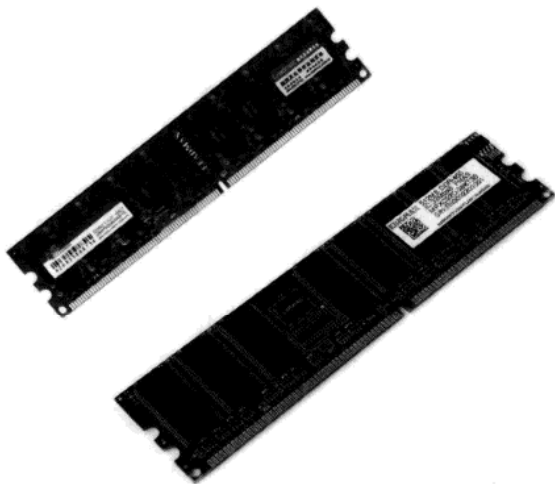


图7.1 内存条

1. 内存的管理方法与地址

CPU 是如何从内存中找到想要的数​​据？想象现实中的仓库，就可以找到答案。如果有一个很大的仓库，为了让所存储的物品容易管理，这个仓库通常就会被划分为若干个房间，而且每个房间都会有一个编号，根据这个编号就可以找到相应的房间，然后就可以找到相应的货物。

内存采取了同样的管理方法，首先内存被划分为若干个很小的存储单元，然后将这些存储单元编号。由于内存中的存储单元太多，所以这些编号往往都是一个非常大的整数，例如 0x00427C5D（这个数为十六进制表示的大整数）。这些用来标识内存中各存储单元的大整数编号就是“地址”。

2. “直接访问”和“间接访问”——酒店找人

再来谈谈去酒店找人的例子。为了找到某人你需要到前台去询问该人的房间号，然后根据房间号找到该人，如图 7.2 所示。有没有一种更简单的方式呢？当然有。如果你已经知道



房间号了,那么就无须再向前台询问了,可以直接去找人。“直接找”当然比“间接找”要快很多。在计算机中也提供了对一个数据的“直接访问”和“间接访问”两种访问方式。



图7.2 到酒店找人

通过变量来访问数据就是间接的访问方式,通过地址来访问数据就是直接的访问方式。

3. 变量、内存存储单元与地址

变量其实就是内存存储单元的一个临时的别名。而地址才是这个存储单元的永久的真名,是不可以更改,也不可以重复的。内存中不会有两个内存存储单元具有相同的名字,但是在程序中同一个变量名可能代表着不同的内存单元。只要变量的作用域不同,程序就允许变量同名。

现实生活也是这样,两个人的名字可能相同,因此每个人还有一个编号来进行区分,在社会上这个编号可以是“身份证号”,在学校里这个编号也可以是“学号”。内存中用来唯一标识和区别内存存储单元的就是“地址”。

计算机内部只能通过地址来找寻内存单元。如果用一个别称(即变量)来标识存储单元,那么计算机就需要先把这个别称翻译成地址,然后去找相应的内存单元,因此这个过程是一个间接访问的过程。就像你去酒店找人,如果不知道房间号就需要去前台询问某个名字的人住在哪个房间;如果你已经知道存储单元的地址了,那么你就可以通过该地址直接去访问存储单元,而无须计算机底层对这个别名进行翻译。也就相当于你去酒店找人前已经知道那个人所在的房间号时就无须再向前台询问了。

4. 指针与指针变量

在C程序设计的时候,程序员先已知一个变量的变量名,然后从计算机那里得到这个变量名对应的内存地址,这个地址就是指针,它指向了一个变量的存储位置,也就是存储该变量的内存单元的地址。因此,在C语言中的指针,更准确地说,应该是指变量的地址。

指针也是一种数据,它当然也可以被存在一个内存存储单元中。如果用一个变量来存放另一个变量的地址,那么它就是一个“指针变量”,即用来存储指针的变量,就称为“指针变量”。指针变量的值就是指针。如图7.3所示,变量p中存储的值是变量j的地址,因此变量p就是一个指针变量。

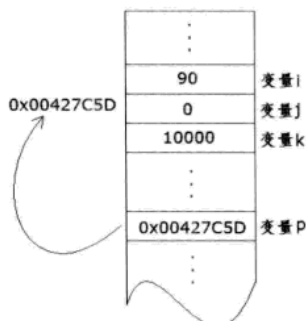


图7.3 指针变量

7.1.2 定义指针变量

1. 指针变量定义

在 C 语言中，变量在使用之前必须被定义，明确其类型，并分配存储空间。指针变量必须依托它所指向的变量，才能具有实际的意义。为了表示指针变量和它所指向的变量之间的关系，C 语言中使用“*”来表示这种指向关系。例如有如下定义：

```
int i = 0;
int *p = i;
```

以上语句表示指针变量 *p* 指向了一个变量 *i*，而 **p* 就表示 *p* 所指向的变量，也就是说 **p* 就表示 *i*。一般的指针变量的定义形式为：

类型说明符 * 指针变量名；

如果希望改变指针所指向的内容，可以采用取地址符“&”，例如：

```
int i = 0;
int j = 0;
int *p = i;
p = &j;
```

上面这段代码首先定义了一个指针变量 *p* 并使它指向变量 *i*，而后又使它改指向 *j*，此过程如图 7.4 所示。

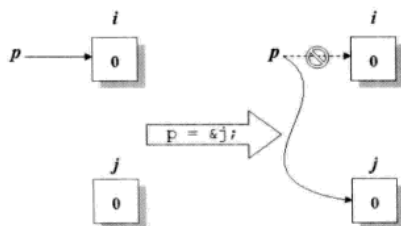


图7.4 改变指针的指向

2. int *是类型名

*int * p* 这个定义方式需要我们深入理解，其中 *int ** 可以作为一种类型名来看待，它表示 *p* 是一个 *int* 型指针。而 **p* 就可以当成一个 *int* 型变量来使用。这种描述方式虽然便于读



者理解，但也常常引起误会。如果 `int *` 可以作为一种类型名来看待，那么下列的定义语句将实现怎样的功能呢？

```
int * a, b, c;
```

相对于类型而言，可以认为“*”和变量名结合的关系更紧密些，所以上述定义的结果是 `a` 是一个指针变量，而 `b` 和 `c` 都是 `int` 型变量。如果想将 `a`、`b` 和 `c` 都定义成指针型变量就只能采取如下的方式：

```
int * a;  
int * b;  
int * c;
```

3. 定义指针时必须指定其类型

不同类型的变量在内存中所占据的空间不同，所以在定义指针时必须指定其类型，这对于指针的移位操作具有很重要的意义。

对于上例中的指针类型 `p`，如果有操作 `p++`，那么对于指针变量 `p` 自加 1 意味着什么呢？因为 `p` 是一个 `int` 型指针，因此 `p++` 操作意味着将指针 `p` 移动 4 个字节（Visual C++ 6.0 中一个 `int` 型变量占据 4 个字节的空间）。可见正确的定义指针类型是非常重要的。如果类型不匹配很可能得到错误的数据或发生访问越界，这些都是非常危险的行为。

4. 指针变量中存放的值能且仅是变量的地址

将一个整数赋给指针变量是错误的。尽管地址本身也是一个大整数，但是前面已经说过指针是变量的地址，这个地址的获得必须通过一个已经被分配了存储空间的变量来完成，变量的地址只能通过指针运算符“*”或取地址符“&”来获得，而不能直接指定一个具体的地址。因此，下面的语句是错误的：

```
int * a;  
a = 0x5050;
```

5. *&p 和 *&p 的区别

因为“*”和“&”具有相同的优先级，按照从右向左的方向结合。对于 `&*p` 而言首先进行一次 `*p` 运算再取地址，就相当于 `&(*p)`。同理，`*&p` 相当于 `*(&p)`，就是先取地址再做“*”运算，`*&p` 等价于 `p`。

6. 指针的优点

前面已经介绍过，指针提供了一种直接访问的方式。直接访问除了更加高效以外，它还能完成一些间接访问无法实现的功能，指针提供了更大的编程灵活性，这种灵活性允许程序员透过层层屏障去操作计算机的底层，这或许是更多 C 程序员对指针爱不释手的更重要的原因所在。

7.1.3 使用指针变量

要想随心所欲地运用指针，首先还是要弄明白指针和变量的关系。下面这段示例代码能够帮助读者理解一些令人困惑的问题：

```
#include <stdio.h>
```

```

int main(void)
{
    //定义整型变量 a 和整型指针 pint
    unsigned int a = 5;
    unsigned int *pint = NULL;

    printf("&a = %p\n a = %d\n", &a, a);
    printf(" &pint = %p\n pint = %p\n &(*pint) = %p\n", &pint, pint, &(*pint));

    //将变量 a 的地址赋值给指针变量 pint
    pint = &a;

    printf("&a = %p\n a = %d\n", &a, a);
    printf(" &pint = %p\n pint = %p\n &(*pint) = %p\n", &pint, pint, &(*pint));

    //打印*pint 的值
    printf(" *pint = %d\n", *pint);

    //改变*pint 的值即是改变变量 a 的值, 反之亦然
    *pint = 10;

    printf("&a = %p\n a = %d\n", &a, a);
    printf(" &pint = %p\n pint = %p\n &(*pint) = %p\n", &pint, pint, &(*pint));
    printf(" *pint = %d\n", *pint);

    return 0;
}

```

编译并运行上述程序, 运行结果如图 7.5 所示。这段程序主要是为了说明整型变量 `a` 及其地址 `&a`、指针变量 `pint` 及其所指对象 `*pint`, 以及指针变量 `pint` 的地址 `&pint` 等表达式之间的错综复杂的关系。如果读者能把这些问题都理顺, 那么就表明读者对指针的认识已经非常到位了。



图7.5 程序运行结果

上述程序研究如下: 程序在最开始定义了一个整型变量 `a`, 并将其赋值为 5。此时, 整型变量 `a` 的值是 5, 变量 `a` 所在内存单元的地址 `&a` 是 0x0012FF7C。程序又定义一个指向整



型变量的指针 `paint`，最开始该指针变量的值为 `NULL`，即它不指向任何整型变量。指针变量 `paint` 被存放在内存的某个存储单元中，它的地址 `&paint` 为 `0x0012FF78`，刚好与 `&a` 相差 4 个字节。

如图 7.6 所示，整型变量 `a` 的值是 5，因为它占据 4 个字节，共计 32 个比特，也就是 32 个二进制位，所以用十六进制数来表示就是 `0x00000005`，其地址 `&a = 0x0012FF7C`。指针变量 `paint` 的值为 `0x00000000`，这表示它不指向任何存储单元，指针变量 `paint` 的地址 `&paint` 是 `0x0012FF78`。

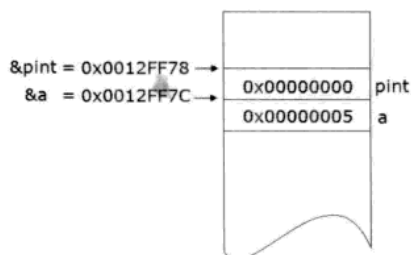



图7.6 变量与地址

 **提示：**因为 Visual C++ 6.0 中整型变量占据 4 个字节的内存空间，而这两个变量（一个整型变量和一个指针变量）在程序中的定义是相连的，因此它们在内存中的存储单元也是相邻的。

前面已经讲过，如果把 `int * paint` 定义方式中的 `int *` 作为一种类型名来看待的话，这个定义就表示 `paint` 是一个 `int` 型指针，而 `*paint` 就可以当成一个 `int` 型变量来使用。而且在定义了一个指针变量 `p` 后，就可以通过 `*p` 来使用指针所指向的对象。

从上述输出结果中可以看到 `paint` 的值始终等于 `&(*paint)` 的值，如何理解表达式 `&(*paint)` 的意义？如果把 `*paint` 看作是一个整型变量，那么 `&(*paint)` 就是对此变量取地址，这个地址也就是指针变量 `paint` 所存储的值，因此 `paint` 始终等于 `&(*paint)`。本来指针变量 `paint` 是不指向任何存储单元的，但是语句 “`paint = &a;`” 将变量 `a` 的地址赋给了指针变量 `paint`，于是整型指针 `paint` 也就指向了变量 `a`。

如图 7.7 所示，指针变量 `paint` 的值变成了 `0x0012FF7C`。此时 `*paint` 表示一个 `int` 型变量，该变量的地址等于 `paint` 的值。在本例中，`paint` 的值就是整型变量 `a` 的地址，所以 `*paint` 就等同于整型变量 `a`，所以它的值等于 5。

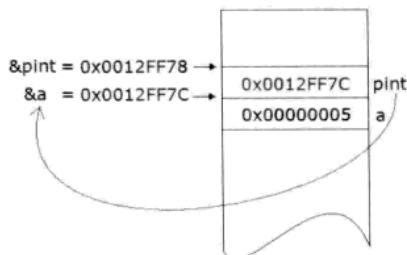


图7.7 给指针变量赋值

由于*pint 等同于整型变量 a, 所以修改*pint 也就相当于修改变量 a。当执行了语句“*pint = 10;” 后, 变量 a 的值就由 5 变为了 10。

7.2 指针与数组

指针与数组之间有着十分紧密的关系。在介绍数组的时候, 细心的读者应该已经留意到了在传递数组参数时, 我们已经使用了指针运算符“*”, 但那时并没有明确地提出指针的概念。本节就将从数组与指针这对密不可分的概念出发, 来向读者详细介绍它们之间的关联。

7.2.1 用指针访问数组元素

任何存储在内存中的变量都有地址, 数组有地址, 数组中的元素也有地址。上一节中我们介绍了关于指针的一些概念, 那么如何使用指针来访问数组中的元素呢? 这一节将就此展开讨论。

1. 定义指向数组的指针

前面的内容告诉读者引用数组中的元素可以使用下标。除了使用下标, 还有一种更经济、更高效的做法就是使用指针。可以定义一个指向数组的指针, 例如:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
int * p;  
p = &a[0];
```

上述代码中首先定义了一个数组 a, 然后定义了一个 int 型指针 p, 并将数组 a 的首地址赋给了 p。前面已经说过, 数组中的元素是按顺序线性摆放的。于是通过对指针 p 进行操作就可以轻松使用数组中的元素了。当然前面的代码也可以改写成如下的形式:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
int * p = a;
```

数组 a 与指针 p 的对应关系如图 7.8 所示。

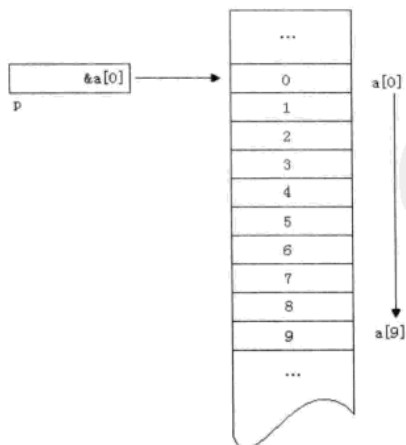


图7.8 数组与指针的关系



下面这段示例程序可以帮助读者进一步理解数组与指针的关系:

```
#include "stdio.h"

void main() {

    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int * p;
    p = &a[0];

    printf("a[0] = %X;\n", a[0]);
    printf("&a[0] = %X;\n", &a[0]);
    printf("&a = %X;\n", &a);
    printf("a = %X;\n", a);
    printf("p = %X;\n", p);
    printf("*p = %X;\n", *p);
}
```

图 7.9 是上述程序的运行结果。可见指针 *p* 中存放着数组 *a* 的第一个元素 *a*[0] 的地址, 数组 *a* 的地址 *&a* 和数组名 *a* 的值都与首元素 *a*[0] 的地址相同。**p* 的内容就是首元素 *a*[0] 的值。

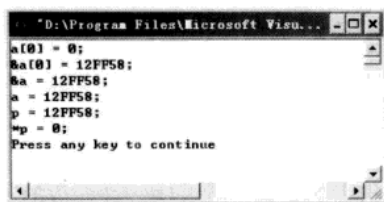


图7.9 数组与指针

2. 通过指针获得其他元素

现在来考虑如何通过指针 *p* 来获得数组 *a* 中的其他元素。

假设目前 *p* 中的值仍为 *&a*[0], 则基于这样的一个事实——数组元素在内存中是顺序地、连续地存储的, 即逻辑上相邻的数组元素在物理地址上也是相邻的, 那么就可以通过将指针 *p* 向上或向下移动来访问数组 *a* 中的任意元素。*p*+*offset* 就是 *a*[*offset*] 的地址, 或者说将指针 *p* 移动偏移量 *offset* 后, 它就指向数组 *a* 中的第 *offset* 个元素。前面的例子告诉我们, 数组名 *a* 的值都与首元素 *a*[0] 的地址相同, 所以 *a*+*offset* 将具有同 *p*+*offset* 一样的意义。

相对地, *(*p*+*offset*)或*(*a*+*offset*)就是 *p*+*offset* 或 *a*+*offset* 所指向的数组元素 *a*[*offset*]。另外, 指向数组的指针变量可以带下标, 所以 *p*[*offset*]与*(*p*+*offset*)等价。例如下面这段示例代码中, 同样先定义了一个数组 *a*, 并让一个指针指向该数组, 然后通过指针操作修改数组中的元素并输出修改结果。

```
#include "stdio.h"

void main() {

    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int * p;
```



```

p = &a[0];

printf("a[5] = %d;\n", a[5]);
*(p+5) = 50;
printf("a[5] = %d;\n", a[5]);
printf("**(p+5) = %d;\n", *(p+5));
}

```

上述代码非常明晰地演示了利用数组访问指针元素的方法。

7.2.2 直接插入排序

本书介绍过冒泡排序法，实际排序算法有很多种，这里就向读者介绍另外一种——直接插入排序法。直接插入排序的基本思想是：数据元素被存储在一个已经有序的序列中，即当插入第 i 个元素 $V[i]$ 时，前面的 $i-1$ 个元素 $V[0]$ 、 $V[1]$ 、...、 $V[i-1]$ 已经排好序，用第 i 个元素的值同已经存在的 $i-1$ 个元素的值从后往前顺序进行比较，找到合适的位置以后就将这 $V[i]$ 插入，而插入点以后的元素都随之向后移动一位。

直接插入排序的过程可以进一步理解为：一个长度为 i 的数组被分为两个集合，如图 7.10 所示，即已排序集合和未排序集合。开始时已排序集合为空，而未排序集合即为整个数组。当排序开始后每插入一个对象，已排序集合元素数目加 1，相应地，未排序集合的元素数目减 1，重复插入过程，直至将未排序集合清空，这时已排序集合就是最终结果。

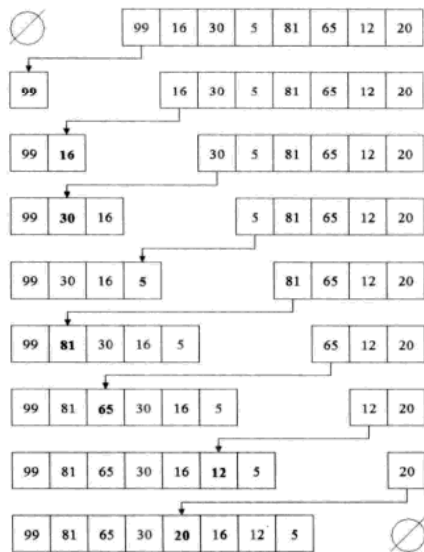


图7.10 直接插入排序

下面给出了实现直接插入排序算法的示例程序，请读者注意其中通过指针来操作数组元素的部分：

```

#include "stdio.h"
#define SIZE 8

```



```
void main()
{
    int a[SIZE] = {99, 16, 30, 5, 81, 65, 12, 20};
    int i, j, k;
    for(i = 1; i < SIZE; i++)
    {
        k = *(a+i);
        j = i-1;
        while(j>=0 && k>*(a+j))
        {
            *(a+j+1) = *(a+j);
            j--;
        }
        *(a+j+1) = k;
    }

    for(i = 0; i < SIZE; i++)
        printf("%d ", *(a+i));
}
```

请读者完成编码后，自行编译并运行程序。

7.2.3 用指针操作多维数组

多维数组是以一维数组为基础扩展而来的，二维数组是最简单的也是最常用的多维数组。这里仍以二维数组为例来说明如何利用指针操作多维数组。

1. 利用指针来操作多维数组实例

通过前面的介绍，读者应该都已经明确这样一个事实，那就是二维数组可以看成是一个特殊的一维数组，假设有一个 3 行 4 列的二维数组 `int a[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}`，那么数组名 `a` 就是该二维数组中第一个元素即 `a00` 的地址。

在内存中二维数组的元素是按行首尾相接线性排列的。所以 `a[3][4]` 二维数组中的各项在内存中的排列是这样的：`a00, a01, a02, a03, a10, a11, a12, a13, a20, a21, a22, a23`。可以试图将其推广到更加复杂和通用的情况下，例如有一个二维数组 `a[M][N]`，若将其转化成一个一维数组 `A[M×N]`，那么原二维数组中的 `a[m][n]` 项就对应一维数组中的 `A[m×N+n]`。

基于上面的认识，可以编写一个简单的示例程序如下：

```
#include "stdio.h"

void main()
{
    int a[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

    int * p;
    p = &a[0][0];

    for(int i = 0; i<12; i++)
        printf(" %d", *p++);
}
```

编译并运行程序，分析结果，读者可以看到二维数组被正确遍历了。当然除了上面这种访问方式以外，C语言还提供了其他手段用以实现对基于指针的二维数组操作。二维数组的数组名表示了数组中首元素的地址，例如上例中的 `a` 就是该数组首元素的地址，不仅如此 `a` 也代表了二维数组中首行元素 (`a00`, `a01`, `a02`, `a03`) 的首地址。

那么 `a+1` 表示的是什么呢？注意 `a+1` 并不表示元素 `a01` 的地址，事实上，它表示的是数组第二行的首地址。这是因为二维数组名是指向行的，而非指向某个元素，因此 `a+1` 的“1”表示的是一行中全部元素所占的字节数。下面这段简单的示例程序证明了这一点：

```
#include "stdio.h"


void main()
{
    int a[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

    int * p;
    p = &a[0][0];

    printf("%p\n", a);
    printf("%p\n", a+1);
}
```

编译并运行上述程序，该程序输出结果如下：

```
0012FF50
0012FF60
```

 **注意：**由于两个数都是十六进制的，所以 $0x0012FF60 - 0x0012FF50 = 0x10$ ，也就是十进制的 16。Visual C++ 6.0 中每个 `int` 型的变量都占据 4 个字节的存储空间，而数组 `a` 中每行包含 4 个元素，所以一行的存储空间就应当是 16。

`&a[0]` 与 `a` 是等价的，它们都代表数组中第一行首元素的地址，即 `&a[0][0]`；同理，`&a[1]` 与 `a+1` 也是等价的，它们都代表数组中第二行首元素的地址，即 `&a[1][0]`……那么每行的其他列元素又如何得到呢？

对于第一行中的元素 `a[0][x]` 来说，在本例中 `x` 是一个介于 0~3 之间的整数，所有的 `a[0][x]` 就构成了一个简单的一维数组 (`a00`, `a01`, `a02`, `a03`)，其实这个一维数组的数组名就是 `a[0]`，`x` 为这个一维数组的下标。因此，`a[0]+0`、`a[0]+1`、`a[0]+2` 和 `a[0]+3` 就可以分别代表该行中各列元素的地址，即 `&a[0][0]`、`&a[0][1]`、`&a[0][2]`、`&a[0][3]`。如图 7.11 所示，它表示二维数组 `a` 中各元素行列的地址计算方法。

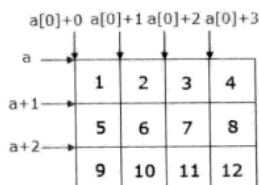


图7.11 二维数组中元素行列地址计算

基于以上讨论的结果再来考虑如何用指针来替换掉数组的下标。对于一维数组而言，`a[0]`



和 $*(a+0)$ 等价、 $a[1]$ 与 $*(a+1)$ 等价,所以可以推出 $a[0]+0$ 就和 $*(a+0)+0$ 等价,它们都代表 $a[0][0]$ 的地址。同理 $a[0]+1$ 也就和 $*(a+0)+1$ 等价,它们都代表 $a[0][1]$ 的地址,所以 $*(a[0]+1)$ 也就能够用来表示 $a[0][1]$ 的值。于是,我们得出了一个结论,那就是二维数组中的某个元素 $a[i][j]$ 可以表示为 $*(a[i] + j)$ 或者 $*(a + i + j)$ 。

2. 应用指针访问二维数组方法的实例

下面举一个简单的例子来帮助读者巩固应用指针访问二维数组的方法。线性代数的知识告诉我们,如果有一个矩阵 A 和一个矩阵 B,那么只有当 A 的列数等于 B 的行数时, $A \times B$ 才有意义,而且矩阵乘法是不满足交换律的。在此前提下,矩阵的惩罚规则是这样的:设 $A=(a_{ij})$ 是一个 $m \times s$ 的矩阵, $B=(b_{ij})$ 是一个 $s \times n$ 的矩阵,那么矩阵 A 和矩阵 B 的乘积是一个 $m \times n$ 的矩阵 $C=(c_{ij})$, 其中:

$$c_{ij} = \sum_{k=1}^s a_{ik} b_{kj} \quad (i=1, 2, \Lambda, m; j=1, 2, \Lambda, n)$$

也就是说,乘积 AB 的第 i 行第 j 列元素是矩阵 A 的第 i 行各元素分别与矩阵 B 的第 j 列各对应元素乘积之和。下面就编码实现两个矩阵 A 和 B 的乘积运算程序。请读者好好理解下面代码中利用指针来操作二维数组的地方:

```
#include "stdio.h"

#define M 3
#define S 2
#define N 4

void main()
{
    int a[M][S] = {3, -1, 0, 3, 1, 0}, b[S][N] = {1, 0, 1, -1, 0, 2, 1, 0};
    int i, j, k, sum, c[M][N];
    int (*p)[N];
    p = c;
    //计算矩阵乘积
    for(i=0; i < M; i++)
        for(j=0; j < N; j++)
        {
            sum = 0;
            for(k=0; k < S; k++)
                sum+=(*(*(a + i) + k)) * (*(b + k) + j));
            *(p[i] + j) = sum;
        }


    //输出计算结果
    for(i=0; i < M; i++)
    {
        for(j=0; j < N; j++)
            printf("%d ", *(c[i] + j));
        printf("\n");
    }
}
```

```

    }
}

```

上述代码中实现矩阵乘法的算法与线性代数中矩阵乘法规则相同，前面已经介绍过，这里就不多解释了。但是读者可能注意到了声明语句“`int (*p)[N];`”，这表示 `p` 是一个指针变量，且它指向一个包含有 4 个整型元素的一维数组。注意由于“`[]`”的优先级更高，因此 `p` 两侧的小括号不能去掉。如果去掉小括号就变成了本章后面将要讲到的指针数组了。

 提示：像语句“`int (*p)[N];`”这样的声明方式可能被用到的地方还有很多，因此请读者掌握这种声明方法。

7.2.4 Z 字形编排过程

通常，在 JPEG 编码过程中，有一个非常重要的步骤，即 Z 字形编排过程。Z 字形编排过程大致是这样的：经过前期处理的图像被分为若干个 8×8 的小图像块，此时就从小图像块的左上角开始沿 Z 字形对图像元素进行遍历，并将遍历所得的结果重新写入等大小的图像块中，整个过程如图 7.12 所示。

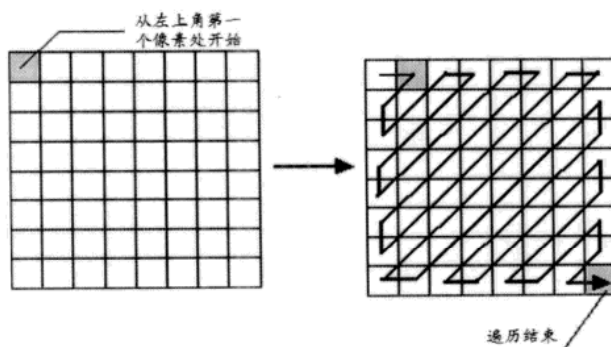


图7.12 Z字形编排方案

经过Z字形排列后，原图像矩阵中的序号如图7.13所示。

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

图7.13 Z字形排列后的图像矩阵示意

要实现这样一个 Z 字形排列可能读者乍一看会感觉无从下手。但是在分析了 Z 字形遍历原矩阵过程中的走向规律，其实可以设计一个非常简单的算法来实现这种编排。对于原始矩阵 `matrix` 中的任意元素 `matrix[i][j]` 的遍历走向规律可以分为如下 3 情况：

- ☐ 如果二维数组中的元素 `matrix[i][j]` 中纵坐标 `j` 是偶数，且 `i=0` 或者 `i=7`，那么遍历路径在矩阵中的走向就是水平向右移动一格。



- ☑ 如果二维数组中的元素 `matrix[i][j]` 中纵坐标 i 是奇数, 且 $j=0$ 或者 $j=7$, 那么遍历路径在矩阵中的走向就是垂直向下移动一格。
- ☑ 除去上述规则以外的情况, 如果二维数组中的元素 `matrix[i][j]` 的横纵坐标和 $i+j$ 是偶数, 那么遍历路径在矩阵中的走向就是右上角移动一格; 若 $i+j$ 是奇数, 则遍历路径在矩阵中的走向就是左下角移动一格。

基于上述原则, 完成矩阵 Z 字形编排功能的代码清单如下:

```
#include "stdio.h"

#define SIZE 8

void main()
{
    int matrix[SIZE][SIZE] = {0};
    int a[SIZE][SIZE] = {0};

    int i, j, x, y, value = 0;

    int * p;
    p = &matrix[0][0];
    //初始化矩阵
    for(i = 0; i < SIZE * SIZE; i++)
        *p++ = i;

    //打印原始矩阵
    printf("原始矩阵如下:\n");
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++)
            printf("%d ", *(matrix + i) + j);
        printf("\n");
    }

    i = 0; j = 0;
    //进行 Z 字编排
    for(x = 0; x < SIZE; x++)
        for(y = 0; y < SIZE; y++)
        {
            (*(a + i) + j) = (*(matrix + x) + y);

            if((i == SIZE-1 || i == 0) && j%2 == 0)
            {
                j++;
                continue;
            }

            if((j == 0 || j == SIZE-1) && i%2 == 1)
            {
                i++;
            }
        }
    }
```



```
        continue;
    }

    if((i+j)%2 == 0)
    {
        i--;
        j++;
    }
    else if((i+j)%2 == 1)
    {
        i++;
        j--;
    }
}

//打印编排结果
printf("\n\n 经过 Z 字编排后的矩阵如下:\n");
for(i = 0; i < SIZE; i++)
{
    for(j = 0; j < SIZE; j++)
        printf("%d ", *(a + i) + j);
    printf("\n");
}
```

请读者完成编码后编译运行程序，并观察输出结果。这个算法不仅对 8×8 的矩阵有效，对于 `SIZE` 取其他值仍然有效，读者不妨试试看。

7.2.5 复杂指针运算的解析

用指针访问数组元素是比较高效的。相对于传统的方法而言，它无须每次都重新计算地址，而使用下标符号“[]”，每次都需要进行变址寻址，这样的时间消耗在数组容量较大时是不可忽略的。特别当采用像 `p++` 这样的操作持续进行偏移访问的时候，效率的提高是很明显的。但是在运用指针变量进行运算时还是要非常小心。

在 C++ 语言中，`++` 运算和 `*` 运算具有相同的优先级，且运算结合的方向是自右向左的，所以 `*p++` 和 `*(p++)` 的作用是完全相同的，即首先得到 `p` 所指向的内容，再将 `p` 向下移动一位（指向下一个元素）。但是 `*p++` 与 `*(++p)` 的作用就不相同了，`*(++p)` 是先对指针 `p` 加 1，再取 `*p`。下面这个例子演示了三者的作用。

```
#include <stdio.h>

void main() {

    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int * p;

    p = &a[0];
    printf("*(p++) = %d\n", *(p++));
    p = &a[0];
```




```
printf("*p++ = %d\n", *p++);  
p = &a[0];  
printf("*(++p) = %d\n", *(++p));  
}
```

程序的运行结果如图 7.14 所示。

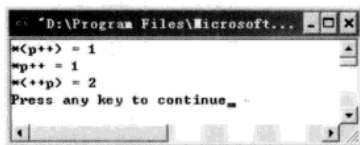



图7.14 指针的运算

$(*p)++$ 表示 p 所指向的内容加 1, 承接上例, 如果 “ $p = \&a[0];$ ”, 且 $a[0]=0$, 则 $(*p)++=1$; 如果 p 当前指向数组 a 中的第 i 个元素, 那么有如下结论:

- ☑ $*(p--)$ 与 $a[i--]$ 的作用相同, 先进行 $*p$ 运算, p 再自减。
- ☑ $*(++p)$ 与 $a[++i]$ 的作用相同, p 先自加, 再进行 $*$ 运算。
- ☑ $*(--p)$ 与 $a[--i]$ 的作用相同, p 先自减, 再进行 $*$ 运算。

在这一点中, 将指针运算和数组名 a 联系了起来。但是两者毕竟存在区别, 并不是所有的运算都一样。特别是可以通过指针运算来改变指针变量的值 (也就是使指针向上或向下移动), 但是不能将同样的操作运用于 a , 即不存在 $a++$ 之类的运算。只是因为 a 表示的是数组首元素的地址, 它是一个指针常量, 所包含的值在程序运行时是不能被改变的。这相当于 “ $\text{int } i = 0; i++;$ ”, 但 “ $\text{int } i = 1++;$ ” 则是错误的。

 **注意:** 与数组存在的问题一样, 指向数组的指针并不会做越界检查, 这是非常危险的, 不慎的访问很可能导致系统的崩溃, 因此要避免这种现象的发生。

7.3 使用字符串指针变量

在上一章中, 我们曾经向读者介绍过字符串的有关知识。在 C 语言中, 字符串是以字符数组的形式来实现的。既然指针可以指向数组, 那么也就表明指针同样可以指向字符串, 因为字符串也是一种数组。本节将对指向字符串的指针进行介绍。

7.3.1 指向字符串的指针

指针也可以用来实现对一个字符数组的操作。这时需要定义一个字符指针, 然后让此字符指针指向一个字符串, 通过该指针即可实现对该字符串的访问。

1. 指针操作字符串的基本方法

下面这段简单的示例程序演示了通过指针操作字符串的基本方法:

```
#include "stdio.h"  
  
void main()  
{  
    char * s = "Good Luck! ";
```

```
    printf("%s\n", s);  
}
```

这里并不需要事先定义一个字符数组，而仅仅定义一个字符指针变量，并用一个字符串常量来初始化它就可以了。C 语言编译器会自动按照字符数组的方式来处理该字符串，而指针变量则指向该字符数组的首元素地址。

例如上例中的变量 `s` 中存储的只能是一个地址，而非是某个具体的字符，更不是整个字符串。语句 “`char *s = "Good Luck!";`” 也可以写为：

```
char * s;  
s = "Good Luck!";
```

但不能写为：

```
char * s;  
*s = "Good Luck!";
```

`printf` 语句中的 `%s` 可以用来输出整个字符串。我们已经知道计算机在遇到字符串结束时知道该字符串已经完结，因此在使用 `%s` 时无须指明字符串的具体长度，计算机也可以正确地将整个字符串输出。

另外，在用 `printf` 语句输出字符串时，其参数 `s` 可以是字符数组的数组名，也可以是指向字符串的一个字符指针变量，这意味着字符数组名或字符指针变量名能够让计算机输出该字符数组中的全部元素。但是，对于其他类型的指针，这种方法是不可行的。例如下面的语句就是错误的：

```
int a[5] = {1, 3, 5, 7, 9};  
printf("%d ", i);
```

在操作字符串的时候，可以使用字符数组的下标，也可以使用指针。其基本方法同一般的指向数组的指针相同。为了帮助读者巩固这部分内容，下面通过一个示例程序来说明指向字符串的指针的使用方法。在上一章中介绍字符数组的时候，我们给出了 C 语言库函数中定义的一些字符串操作函数。

2. 字符串操作函数实例

下面这段示例程序中重写了其中 3 个比较常用的字符串操作函数，分别是字符串复制函数、字符串比较函数和字符串连接函数。这些示例程序非常鲜明地揭示了利用指针操作字符串的一般方法，请读者通过示例程序仔细体会。

```
#include "stdio.h"  
  
//字符串复制  
char * strcpy1(char * p1, char * p2)  
{  
    while((*p1++ = *p2++) != '\0');  
    return p1;  
}  
  
//字符串复制的另一种等价实现  
char * strcpy2(char * p1, char * p2)  
{
```



```
while((*p1 = *p2)!='\0')
{
    p1++;
    p2++;
}
return p1;
}

//字符串比较
int strcmp1(char * p1, char * p2)
{
    for( ; *p1==*p2; )
    {
        if(!*p2)
            return 0;
        p1++;
        p2++;
    }
    return(*p1-*p2);
}

//字符串连接
void strcat1(char * p1, char * p2)
{
    char *p = p1;
    while(*p1)
        p1++;
    while(*p2)
    {
        *p1=*p2;
        p1++;
        p2++;
    }
    *p1='\0';
    p1 = p;
}

void main()
{
    char * s2 = "To be or not to be, that is the question!";
    char s1[45] = {'a'};
    strcpy1(s1, s2);
    printf("%s\n", s1);

    char * s3 = "china";
    char * s4 = "china";
    char * s5 = "chinese";

    if(strcmp1(s3, s4)==0)
        printf("They are the same!\n");
}
```




可以看到当需要将字符串作为参数进行传递时,如果使用指针形式作为参数,在函数内部操作的方法就是通过指针的移动来访问逐个字符。当然,除了指针形式以外,用字符数组作为参数进行传递也是可以的,在函数内部操作指针或是通过下标来访问都是被允许的。这一点与第6章所讲的内容无异,这里就不再赘述了。

7.3.2 与字符数组的比较

在第6章介绍字符串的时候,我们就已经说过C语言中的字符串是以字符数组的形式实现的。而本章学习了指针之后,我们又知道通过字符指针变量是可以操作和访问字符数组的。再加上数组与指针之前有着非常紧密的联系,因此很多初学者往往容易把它们混为一谈。将字符数组和字符指针变量融会贯通,并强调它们之间的联系能够帮助程序员更自由地开发程序,更灵活地编写代码。

但是在做到融会贯通之前,仍然还需要把它们分开来看,既掌握它们的联系,更要注意它们的区别。字符数组与指向字符串的字符指针变量最本质的区别就在于字符数组是确实实实在在地由若干个字符元素所组成,一旦字符数组被定义,它的每个元素就会在内存中占据一定的实际空间。而字符指针变量中存放的是一个地址,妄图将字符串存储到字符指针变量中是不可能的。正因为二者之间有着这样本质的区别,因此在实际使用的时候也会产生一些差异。这些差异可以总结为以下几点。

1. 赋值方式的不同

在对字符数组进行赋值的时候,不可以对整个字符串进行赋值,而只能对其中的某个字符进行赋值,因此下面的语句是错误的:

```
char str1[12], str2[12], str3[12];
str = "Good Luck!";           ///Wrong!!!
str1[] = "Good Luck!";        ///Wrong!!!
str2[12] = "Good Luck!";      ///Wrong!!!
```

但是对于字符指针变量来说,却可以采用下面的赋值方法:

```
char * str;
str = "Good Luck!";           ///Right!!!
```

上述赋值方法也可以写为:

```
char * str = "Good Luck!";    ///Right!!!
```

在第6章中我们已经学过,对字符串赋值时可以采取下面的方法:

```
char str[12] = "Good Luck!";  ///Right!!!
```

总之字符数组在定义时是可以整体赋初值的,但是在赋值语句中就不能对字符数组进行整体赋值了。而指向字符串的指针不仅可以在定义时整体赋初值,在赋值语句中也可以对其进行整体赋值。但是要注意,这些值(也就是一组字符)并不是赋给指针变量的。这些赋值语句的意思只是让指针所指向的字符数组中的内容被改变,而非改变指针本身。

2. 地址是否可改变

指向字符数组的指针变量和字符数组名本质上都表示数组的首地址。但是指针可以通过一定的计算(例如自增或者自减)来改变其自身的指向,而数组名却不可以。看下面这段示例程序:

```
#include "stdio.h"

void main(){

    char * str = "Good Luck!";
    str += 5;
    printf("%s\n", str);
}
```

编译并运行上述程序，输出结果如下：

Luck!

很容易理解，输出结果是由于指针 `str` 的指向向前移动了 5 个字符长度而造成的。然而，这样的运算对于数组名是不可以进行的，因此下面的示例程序是错误的：

```
#include "stdio.h"

void main(){

    char str[] = "Good Luck!";
    str += 5;        //Wrong!!!
    printf("%s\n", str);
}
```

3. 杜绝指针空指向


本部分要介绍的问题归根结底仍然是由于字符数组同字符指针变量二者的本质区别所导致的，但是为了说明这种影响，还是请读者来看一段示例代码：

```
#include <stdio.h>

void main()
{
    char * str;
    scanf("%s", str);
}
```

读者是否意识到了上面这段程序有什么不对劲的地方吗？不妨在计算机上运行一下试试看。笔者在 Visual C++ 6.0 下编译并运行了该程序，在编译过程中，编译器抛出了一个警告，我们先忽略它。

程序开始运行后，我们随机从键盘上输入一个字符串后并按下回车键，结果程序崩溃了，计算机弹出了一个错误对话框，如图 7.16 所示。

 **注意：**在不同的环境下可能得到的结果不一定相同，但这就意味着风险。

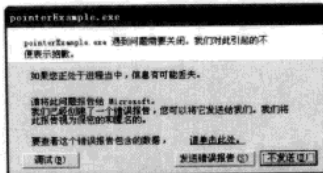


图7.16 错误对话框




为什么会出现上述的错误？这是一个值得我们思考的问题。为了探究问题的原委，我们采取了另外一种方法来实现上述程序，请读者来看下面这段代码：

```
#include <stdio.h>

void main()
{
    char str[10];
    scanf("%s", str);
}
```

编译并运行程序后，一切正常。细心的读者可能已经从对比中看出问题的所在了。如果程序中定义了一个字符数组，编译器在编译时就会为这个数组分配内存单元，数组中的每个元素都有确定的地址。但是定义一个字符指针变量时，编译器只会给这个指针变量分配存储空间，而不会为其所指向的字符变量分配空间。如果将这个指针指向一个具体的字符变量或者字符数组，这个指针变量才可以获得一个有意义的值。否则，它就只能处于一种“悬空”的状态。

 **提示：**所谓“悬空”的状态就是指针指向了一个地方，但是这个地方是任意的，是没有具体意义的，也是程序员无法控制的。如果对这样一个指针进行操作，结果会导致一些本来可能不该被访问的地方意外地被修改了。这是一种极其危险的行为，所以程序就崩溃了。为了避免这种不安全情况的出现，通常的规范是一旦声明了一个指针，就要让它指向一个有意义的存储单元。

例如在下面这段示例程序中，字符变量 `p` 指向了一个已经存在的字符数组 `str`，这样再使用指针 `p` 就非常安全了。

```
#include <stdio.h>

void main()
{
    char * p, str[10];
    p = str;
    scanf("%s", p);
}
```

4. 格式化的字符串

字符指针变量和字符数组都可以用来表示一个格式字符串。通过使用不同的格式字符串去替换 `printf` 语句中原有的格式字符串就可以动态地改变输入/输出的格式。例如下面这段示例代码：

```
#include "stdio.h"

void main(){

    int a = 100;
    double b = 3.14;
    char * str = "a = %d, b = %.2f\n";
```



```
printf(str, a, b);  
}
```

上述代码也可以改写为:

```
#include "stdio.h"  
  
void main(){  
  
    int a = 100;  
    double b = 3.14;  
    char str[] = "a = %d, b = %.2f\n";  
    printf(str, a, b);  
}
```

它们的作用都相当于下面这段代码:

```
#include "stdio.h"  
  
void main(){  
  
    int a = 100;  
    double b = 3.14;  
    printf("a = %d, b = %.2f\n", a, b);  
}
```

但是由于不能对字符数组进行整体赋值,所以使用指针变量来指向不同字符串的方式就显得更为灵活方便了。

7.3.3 如何输出其自身的程序

指针的妙用真是说不完,我们一同来看一段程序,注意程序语句的非断行处是没有回车换行的:

```
#include <stdio.h>  
char*f="#include <stdio.h>%cchar*f=%c%s%c;%cint main(){printf(f,10,34,f,34,10,10);return 0;}%c";  
int main(){printf(f,10,34,f,34,10,10);return 0;}
```

你能猜到这段程序到底都做了些什么工作?不妨在 Visual C++ 6.0 中尝试运行一下,图 7.17 给出了运行结果。

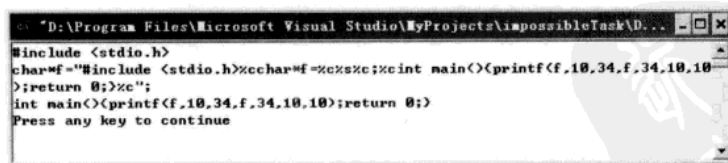


图7.17 程序运行结果

读者发现其中的奥妙了吗?没错!这段程序的运行结果居然就是输出它自己!很难想象,这种程序不仅仅是笔者所列出的这段,事实上,它是一类程序。

很多程序设计爱好者还专门研究了一种程序的设计方法。就上面这段示例程序而言,main()函数中仅仅执行了一条 printf 语句。而该 printf 语句把一个指向字符数组的指针变量作



为格式化字符串来使用。

除此之外，这类程序的设计还要基于以下两个最基本的原理：

- ☑ 数据在计算机中的表示，读者不难发现程序中频繁地出现 10 和 34 这样的数字。这是因为 10 在 ASCII 码中刚好表示换行，而 34 则在 ASCII 码中表示双引号。如果在这类程序中出现类似的数字，其作用基本相同。
- ☑ 所有这些程序的实现都必须借助指针，没有指针这类程序就不存在！在使用指针的时候，程序员必须时刻清楚指针所指向的东西是什么。在本节所涉及的问题上，读者应该明确上述示例程序中所使用的指针指向的是一个格式化字符串。

随着对 C 语言学习的深入，相信读者对这种“变态”程序的理解将会得到进一步加深。如果读者自己也能够写出这样一个输出自身的程序，那么也就说明了对于前面两个关键问题你已经有了较为深刻的认识。理解代码的乐趣也在于此。因为理解代码，所以代码就会被赋予魔力，因为理解代码，代码就能使不可能变为可能！

7.4 指针与函数

将指针与函数联系起来是一件非常有趣的事情。C 语言中不仅可以让函数返回指针类型的数据，也可以让指针用作函数的参数来进行传递。而且事实上，指针跟函数的关联还不仅如此，C 语言中甚至允许有指向函数的指针。本节就来谈谈指向函数与指针的一些问题。

7.4.1 将指针用作函数参数

前面我们已经对函数的相关内容进行了介绍，因此读者应当明确 C 语言中函数传递采用的是一种被称为“按值传递”的方式。当使用按值传递方式时，函数内部将对参数做一个副本，而函数只是对此副本进行操作，而不改变原值。

按值传递是 C 语言中使用的一种默认传值机制。这种机制的特点在于当被传递的参数值在函数体内发生改变时，调用主体中的原始数值不会发生改变。

1. 按值传递的实例

下面就通过一个例子来帮助读者回忆一下按值传递的特点：

```
#include "stdio.h"

void fun(int x)
{
    printf("x = %d\n", x);
    x++;
    printf("x = %d\n", x);
}

void main()
{
    int x = 0;
    printf("x = %d\n", x);
    fun(x);
}
```

```
printf("x = %d\n", x);  
}
```

运行上述程序，结果如图 7.18 所示。因为函数 fun() 采用了按值传递的参数传递方式，因此在函数体内生成了参数 x 的一个副本，对这个副本的值进行自加运算，于是这个副本的值由 0 变到了 1。一旦函数 fun 返回，那么这个副本的值的生命周期就结束了，返回主调函数后，原变量 x 的值并没有被改变，于是程序输出了 0。

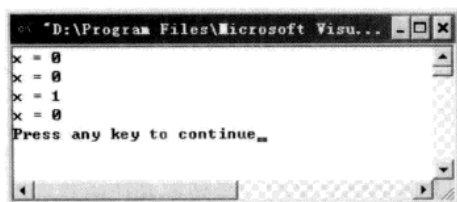


图7.18 按值传递程序运行结果

在某些需求下，程序员希望参数的结果能够在函数返回后仍然被保留下来，这时就需要使用指针。在上述程序中，函数 fun() 中的形参变量 x 与 main() 函数中的实参变量 x 是两个完全不同的变量，因为它们被分别存储在不同的空间中。

如果希望函数的形参被修改的同时，其主调函数中的实参也随之变化，那么使用指针传递的方式不失为一个好的选择。既然已经知道按值传递仅仅是在函数体内部对参数的一个副本进行操作而无法改变外部变量，那么就很容易想到只要获得所传参数的指针，然后直接对指针所指向的值进行修改就可以满足要求了。

2. 指针传递实例

当进行函数参数传递时不是传递一个普通变量而是传递一个指针变量，这就是指针传递方式。来看下面的例子：

```
#include "stdio.h"  
  
void fun(int* p)  
{  
    printf("*p = %d\n", *p);  
    (*p)++;  
    printf("*p = %d\n", *p);  
}  
  
void main()  
{  
    int x = 0;  
    printf("x = %d\n", x);  
    fun(&x);  
    printf("x = %d\n", x);  
}
```

编译并运行上述程序，运行结果如图 7.19 所示。可见外部变量的值被成功地修改了。

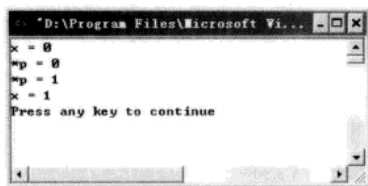


图7.19 指针传递程序运行结果

指针传递在 C 程序设计实践中非常常见，其好处为：

- ☐ 能让函数形参的变化映射到实参上，从而在函数调用结束后仍然保留参数变化的痕迹。
- ☐ 提供了一种高效的传值方式。

如果被传递的参数是一个很大的对象，例如一个结构体（结构体类型将在本书的后面具体介绍），按值传递的方式会复制该对象的一个副本并在函数内操作这个副本。复制大对象的过程本来就很浪费时间，如果只是传递一个指针的话，效率就高多了，因为不用再为大对象复制而烦恼。指针不过是一个大整数，而大整数要远比大对象小得多。因此使用指针传递方式是非常高效的。

7.4.2 指向函数的指针

计算机中正在运行的程序（无论是数据还是指令）都被存储在内存中。那么这样说的话就揭示了一个惊人的事实——既然变量都有地址，那么指令也应该有地址。的确如此，指令也是有地址的。那么指针能不能被用来指向程序中的任意一条指令？C 语言中并没提供这样的语法。但是 C 语言中却提供了另一种具有一定限制的指令访问方式——指针变量可以指向一个函数。

指向函数的指针变量所包含的地址其实就是函数中第一条指令的地址，也就是该函数的指针。调用一个函数时除了通过函数名来调用它以外，还可以通过指向该函数的指针变量来调用它。一个指向函数的指针其初始值不能为空，因为它在使用之前必须被赋予一个真实的函数地址。指向函数的指针变量的一般定义形式如下，其中的函数类型是指函数返回值的类型：

函数类型 (*指针变量名) ();

请看下面这段示例代码，它使用普通的函数名方式来实现函数的调用，此函数用于实现

矩形法求解函数 $\int_a^b x^2 dx$ 定积分的功能。

```
#include "stdio.h"
#include "math.h"

//intergal(x^2)
double func1(double a , double b)
{
    double sum = 0.0;
    double length = 0.000001;
    double x = a;
    while(x < b)
    {
```

```
        sum += x*x*0.000001;

        x+=0.000001;
    }

    return sum;
}

void main()
{
    double result = 0.0;
    result = func1(0.0, 1.0);
    printf("%g\n", result);
}
```

现在来改写上面的代码，使用一个指向函数的指针变量来调用函数：

```
#include "stdio.h"
#include "math.h"

//intergal(x^2)
double func1(double a , double b)
{
    double sum = 0.0;
    double length = 0.000001;
    double x = a;
    while(x < b)
    {
        sum += x*x*0.000001;

        x+=0.000001;
    }

    return sum;
}

void main()
{
    double result = 0.0;

    double (*p)(double, double);
    p = func1;
    result = (*p)(0.0, 1.0);

    printf("%g\n", result);
}
```

从这段使用指向函数的指针实例中，其实可以获得很多的信息。下面就来逐条解析：

☐ 语句 “double (*p)(double,double);” 定义变量 p 为一个指向函数的指针变量，此函数的返回值类型为 double 类型。

因为 double (*p)(double, double)并非指向某一固定函数，它仅仅是表示定义了这样一个




类型的变量，因此在程序中可以将不同的函数地址赋给它，由此它所指向的函数就会随之变化。该语句中 `*p` 两侧的括号是不能省略的，这样的语法意味着 `p` 先与 “`*`” 结合，是一个指针变量，再与后面的括号 “`()`” 结合，表示此指针变量指向函数而非变量。

如果将 `*p` 两侧的括号去掉，则变成 `double *p()`，表示函数 `p()` 的返回值类型是一个指向 `double` 型变量的指针。因为 “`()`” 的优先级高于 “`*`”，所以 `p` 会先与 “`()`” 结合，由此就声明了一个函数而非指针。


❑ 赋值语句 “`p = func1;`” 将函数 `func1()` 的入口地址（即函数中首条指令的地址）赋给了指针变量 `p`。

可见，在给函数指针变量赋值时，只需要给出函数名即可，并不需要给出参数，因此，如果将上面的赋值语句改写成 “`p = func1(0.0, 1.0);`” 是错误的。但是由于这里仅仅使用了函数名，而不带括号和参数，为了不让编译器将其与变量混淆，必须在使用之前进行声明，表明 `func1` 是函数名而非变量名，这样在编译时它们才会被当做函数名来处理。

 **注意：**在使用函数指针时，只需将 `(*p)` 替代函数名即可，但是需要在其后的括号里显式地添加实参，即使函数不传递任何参数，该括号也不可以省略。

7.4.3 指针对于指令的访问是受限制的

数组名可以代表数组的起始地址，因此函数名也可以代表函数的入口地址（函数中的首条指令的地址），但是对于指向函数的指针变量，它只能指向函数的入口处而无法指向函数中的某一条具体的指令，因此 `p+n`、`p++` 等指针运算对于指向函数的指针是没有意义的。所以我们也称 C 语言中提供的指针对于指令的访问是受限制的。

 **注意：**获得一个函数地址的方法同获得一个变量地址的方法一样。所以前面程序中的语句 “`p = func1;`” 也可以写为 “`p = &func1;`”。

7.4.4 使用指向函数指针的语法来实现编程

指向函数的指针最常用的地方是：

❑ 作为参数传递给其他函数。

❑ 作为参数以实现函数地址的传递，也就是将函数名传递给形式参数。

但是，在某个函数中调用另外一个函数仅仅需要在此函数中直接调用所需函数就可以了，这是 C 语言所支持的非常基本的函数调用语法。如此看来，将函数指针作为参数来传递，再在函数体中使用实在是多此一举、画蛇添足。

然而，事实上将函数指针作为参数来使用并非百无一用，尤其当每次函数所调用的其他函数无法固定时，就显得尤为重要了。假设有函数 `fun()`，在某次执行过程中需要调用函数 `func1()`，而下一次就需要调用函数 `func2()`，再下一次又可能调用 `func3()`。如果使用函数指针，则不必对函数 `fun()` 进行修改，只需要在每次调用函数时通过不同的函数名作为形参传递即可。

这种方法极大地增加了函数使用的灵活性，开发人员可以编写一个通用的函数来实现各种专用的功能，这是符合结构化程序设计思想的方法。因此在实际开发中是值得推荐和提倡的。现在就通过一个简单的示例程序来说明这种方法的应用。

下述示例程序中使用了一个求定积分的通用函数，用它可以分别求得 3 个函数的定积分：

$\int_a^b x^2 dx$ 、 $\int_a^b \sin x dx$ 和 $\int_a^b e^{\sqrt{x}} dx$ 。

可见,每次需要求定积分的函数并不相同,但是我们可以使用一个通用的函数 `intergal()`,并通过 3 个形式参数: 上限 `b`、下限 `a` 和指向函数的指针变量 `fun` 来显示各自专用的定积分求解函数。其中:

☐ 函数 `func1()` 用于求解函数 $\int_a^b x^2 dx$ 的定积分。

☐ `func2()` 用于求解函数 $\int_a^b \sin x dx$ 的定积分。

☐ `func3()` 用于求解函数 $\int_a^b e^{\sqrt{x}} dx$ 的定积分。

以下给出了完整的程序代码清单:

```
#include "stdio.h"
#include "math.h"

//intergal(x^2)
double func1(double a, double b)
{
    double sum = 0.0;
    double length = 0.000001;
    double x = a;
    while(x < b)
    {
        sum += x*x*0.000001;

        x+=0.000001;
    }
    return sum;
}

//intergal(sinx)
double func2(double a, double b)
{
    double sum = 0.0;
    double length = 0.000001;
    double x = a;
    while(x < b)
    {
        sum += sin(x)*0.000001;

        x+=0.000001;
    }
    return sum;
}

//intergal(e^(x^2-2))
```




```
double func3(double a , double b)
{
    double sum = 0.0;
    double length = 0.000001;
    double x = a;
    while(x < b)
    {
        sum += exp(sqrt(x))*0.000001;
        x+=0.000001;
    }
    return sum;
}

void intergal(double a, double b, double (*fun)(double, double))
{
    double result = 0.0;
    result = (*fun)(a, b);
    printf("%g\n", result);
}

void main()
{
    printf("计算 x^2 在 0~1 上的定积分结果: ");
    integral(0.0, 1.0, func1);
    printf("计算 sinx 在 0~ $\pi$  上的定积分结果: ");
    integral(0.0, 3.141593, func2);
    printf("计算 e^(x^2) 在 0~1 上的定积分结果: ");
    integral(0.0, 1.0, func3);
}
```

使用指向函数的指针不仅仅是一种语法上的应用,更重要的是它所体现出来的程序设计思想,即使用一个通用的函数来实现各种专用功能的结构化程序的设计思想。这一点在面向对象的 C++ 和 Java 等语言中得到了淋漓尽致的发挥。面向对象的一个重要特点就是“多态”。多态的内涵正是使用一个通用的形式来完成各种专用的功能。

可见在 C 的世界里能够使用指向函数指针的语法来实现这种编程目的是一种多么先进的思想啊。尽管本书并没有涉及 C++ 或者面向对象的有关内容,但我们仍然希望能够把一些先进的思想传递给读者。待日后读者学习 C++ 时,若能再想到这里所谈到的内容,必定会有更深刻的感悟。

7.4.5 返回值为指针的函数

1. 返回指针值的函数应用

在某些情况下,程序员可能希望函数的返回值是一个指针。这样做最明显的好处就是允许主调函数获得多个返回值,或者更准确地说是一种值。在介绍函数的时候我们曾经讲过,函数的返回值是唯一的。如果想让函数返回更多的值可以使用的方法有很多:

- ☐ 定义全局变量并将希望函数返回的值存储在全局变量里。

☑ 把指针当做参数传进函数，即使函数返回指针所指向的地方仍然保留了原值，这样也起到了获得多个结果的作用。

本节我们要介绍的是另外一种新的方法，也就是让函数返回一个指针。由于指针可以指向数组，因此得到了返回指针，其实也就得到了一组值。

C语言中带返回指针值的函数声明形式如下：

类型标识符 * 函数名(参数列表)；

为了帮助读者理解返回指针的函数的具体应用，下面我们就举一个简单的例子。假设现在有一份成绩单如表 7.1 所示。其中横向上是科目，纵向上是参加考试人的学号。现在希望通过编程来实现这样一个功能，即当输入该学生的学号时将其所有成绩显示在屏幕上。

表 7.1 某班学生的成绩单

	Math	English	History
0	90	85	72
1	100	80	75
2	80	93	98
3	60	53	75
4	78	82	39

下面给出了如何利用指针函数来实现上述功能的代码清单：

```
#include "stdio.h"

#define M 5
#define N 3

float * find(float (*score)[N], int n)
{
    float * p;
    p = *(score + n);
    return p;
}

void main()
{
    float score[M][N] = {{90, 85, 72},
                        {100, 80, 75},
                        {80, 93, 98},
                        {60, 53, 75},
                        {78, 82, 39}};

    int num;
    printf("Please input the number of student:\n");
    scanf("%d", &num);

    printf("His/Her scores are as belwo:\n");
    float * p = find(score, num);
    for(int i = 0; i < N; i++)
```



```
        printf("%.2f\t", *(p+i));  
    }
```

main()函数中定义了一个二维数组 score 来存储这些学生的成绩单。然后函数 find()以该二维数组和用户输入的学号 num 为实参进行传值。在函数 find()内部定义了一个指针 p, 它指向二维数组 score 的第 n 行元素的首地址, 然后指针 p 被返回。以返回的指针 p 为首地址, 通过修改*(p+i)中变量 i 就可将某人的成绩全部输出。

使用返回指针的函数其语法并不复杂:

- ☑ 一方面需要理解函数返回值的意义。
- ☑ 一方需要注意指针的使用。

基于这两点就能够很好地使用指针函数了。

2. 返回指针值函数的错误应用

使用指针函数仍然有一些问题需要读者注意, 在更多的时候其实它并不像看起来的那样简单。如果不小心误用了它, 那么后果可能会糟糕。依然以前面的问题为例, 现在我们想求得该次考试中每个人的平均成绩并将其输出。请读者仔细分析如下代码, 看看它是否可以实现我们的要求。

```
#include "stdio.h"  
  
#define M 5  
#define N 3  
  
float * mean_byPerson(float (* score)[N])  
{  
    float sum = 0.0;  
    float r[M];  
    float * p = &r[0];  
    int i = 0, j;  
    for(; i < M; i++)  
    {  
        for(j = 0; j < N; j++)  
            sum += *(score[i]+j);  
        r[i] = sum/(float)N;  
        sum = 0.0;  
    }  
  
    return p;  
}  
  
void main()  
{  
    float score[M][N] = {{90, 85, 72},  
                          {100, 80, 75},  
                          {80, 93, 98},  
                          {60, 53, 75},  
                          {78, 82, 39}};
```

```
float * m1 = mean_byPerson(score);

printf("Everyone's mean score is as below:\n");

for(int i = 0; i < 5; i++)
    printf("%.2f\t", *(m1+i));
}
```


观察上述程序，main()函数中仍然使用二维数组 score 来存储考试成绩单，函数 mean_byPerson()以此二维矩阵为参数向函数内部传值。在该函数内部，我们定义了一个一维数组 r，并将参加考试的每个人的平均成绩存储到这个一维数组中。指针 p 指向了这个数组，最后函数将指针 p 返回。

通过指针 p 能否得到算出的一组平均成绩呢？请读者编译并运行上述程序，结果表明程序输出了一些莫名其妙的值，也就是说程序出错了。那错误到底从何而起？请注意数组 r 仅仅是函数中的一个局部变量，一旦函数返回，它所占用的存储空间就会被释放。因此，即使保留一个指向 r 的指针 p，也不可能得到数组 r 中原有的值。

函数结束后，p 所指向的空间就已经失控了，我们也不知道程序到底在那块存储空间里放了什么东西，如果程序对 p 进行了操作，那么最直接的结果就是修改了本不该被修改的值。这种行为的后果极其严重：

☑ 一方面可能引起程序崩溃。


☑ 一方面如果程序没有崩溃，会接收一些错误的值然后照常运行，结果就是得到一组非常离奇的输出，但是对于这种情况我们可能很难发现错误到底出在哪里。

 **注意：**通过上面的例子我们需要明确的一点就是函数内部生产的变量会在函数结束后被销毁，因此如果函数返回一个指向局部变量的指针是没有意义的，而且还很危险。

本小节的第一段程序中函数尽管也返回了指针，但该指针所指向的却是主调函数中的变量，因此这样的做法是可以的。对于求每个人成绩平均值的问题，应当如何解决呢？

☑ 一种可选择的方式：在函数内部使用 malloc 来分配一定的存储空间。尽管这个方案可行，但使用起来很麻烦，要注意和顾及的地方很多。

☑ 另一种方式（推荐）：主调函数中定义一个数组，然后将指向该数组的指针传进被调函数中。被调函数可以通过指针修改这个数组，而且当被调函数返回后，这个数组也不会被释放。用这样一个数组来记录一组值并能够实现在函数返回后仍然有效的目的，显然已经满足我们的要求了。这种方法的基本思路我们已经描述的非常清楚了，有兴趣的读者不妨自己试着动手编写一个这样的函数来完成这个要求。

 **说明：**从上述的分析中可以看出，尽管指针函数也可以起到返回一组值的作用，但是它的使用是存在局限性的，即返回的指针所指向的值不能是函数中的局部变量。这一点请读者务必注意。



7.5 复合多维指针的使用

指针不仅可以指向普通变量，还可以指向数组或者指针，这样就形成了一种多维指针，看上去就有点像多维数组。本节就对复合多维指针进行介绍。

7.5.1 指针数组的使用

1. 指针数组定义

数组中的元素可以是任意类型的，例如整型、浮点型或者字符型。数组中的元素当然也可以是指针类型的。如果一个数组中的元素是指针类型的，那么这个数组就是一个指针数组。

还记得数组的两个属性吗？首先数组是若干个元素的有序集合，其次数组中的元素必须都是同类的。指针数组就是一组有序的指针的集合。另外，指针数组里的指针也都必须是同一类型的，例如都是指向整型数据的指针，或者都是指向字符型数据的指针等，这才能形成所谓的“物以类聚”。指针数组的一般声明形式如下：

```
类型说明符 * 数组名[数组长度];
```

其中类型说明符为指针值所指向的变量的类型。例如：

```
int *p[3];
```

上述语句表示 p 是一个指针数组，它由 3 个数组元素组成，每个元素值都是一个指向整型变量的指针。对于这个声明的理解如果能够联系其运算符的优先级会更容易些。因为“[]”的优先级高于“*”，所以 p 先和“[]”结合。显然， $p[3]$ 是一个数组的形式。另外在把 int^* 看成是指向整型变量的指针，所以上述声明语句的意思就是声明一个指针数组，且该数组中包含有 3 个指向整型变量的指针。

2. 指针数组两个典型的用途

☑ 可以用来指向一个二维数组。

我们知道指针是可以指向数组的，那么如果一个指针数组中的每个元素都是一个指向一维数组的指针，不就恰好可以表示一个二维数组了吗？指向数组的指针其实就是数组中第一个元素的地址，所以用来表示二维数组的指针数组中每个元素就将是二维数组中每一行的首元素地址。二维数组与其对应的指针数组关系如图 7.20 所示。

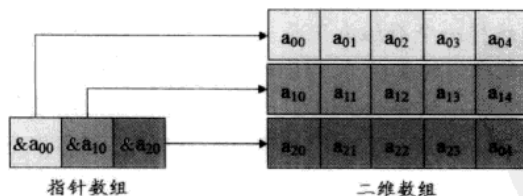


图7.20 表示二维数组的指针数组

下面这段示例代码演示了使用指针数组访问二维数组的方法：

```
#include "stdio.h"

void main()
```

```

{
    int a[3][5]={1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
    int *p[5] = {a[0],a[1],a[2]};


    int i = 0, j;
    //遍历二维数组
    for(; i < 3; i++)
        for(j = 0; j < 5; j++)
            printf("%d ", *(p[i]+j));
}

```

在上述程序中, `p` 是一个指针数组, 数组中的 3 个元素分别指向二维数组 `a` 的各行。`*p[i]` 表示第 `i` 行第 0 列元素值, `*(p[i]+j)` 表示第 `i` 行第 `j` 列的值。

读者注意指针数组和二维数组指针变量的区别。这两者虽然都可用来表示二维数组, 但是其表示方法和意义却是有区别的。指向二维数组的指针变量是单个的变量, 而指针数组表示的是由多个指针组成的一个有序的指针序列。例如下面的语句表示一个指向二维数组的指针变量, 该二维数组的列数为 3。

```
int (*p)[3];
```

 注意: 语句中的小括号是不能省略的。

而下面的语句则表示 `p` 是一个指针数组, 数组中包含有 3 个下标变量 `p[0]`、`p[1]` 和 `p[2]`, 这 3 个变量均为指针变量。

```
int *p[3];
```

☑ 用来指向一组字符串。

前面我们已经向读者讲过了指针是可以指向字符串的, 如果将一组指向字符串的指针形成一个数组, 这时指针数组中的每个元素就将被赋予一个字符串的首地址。这样通过操作数组的元素就可以轻松使用一组字符串了。例如, 现在班上有几名学生, 我们要做的事情是用一个数组来存放他们的姓名。并希望他们的名字在数组中按照字母表的顺序排列。

有的读者可能会想到用二维字符数组来实现, 但是在定义二维数组的时候, 列数必须被明确给出, 而且二维数组中每一行所包含的元素个数也都必须相等。但是人名总是有长有短, 如果用二维数组来实现难免会浪费宝贵的存储空间。如果用指针数组来实现就方便多了, 人名与数组的对应关系如图 7.21 所示。

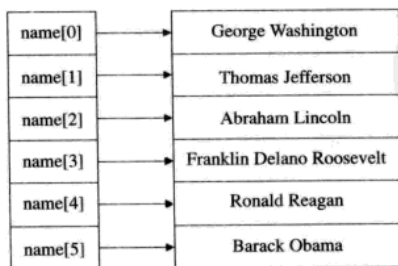


图7.21 通过指针数组来操作一组字符串

如果使用指针数组来存储指向字符串的指针:



- ☑ 各字符串在数组内的位置调整将更加方便。这时只需要改变数组内各指针的指向，而无需实际调整字符串在内存中的存放位置。
- ☑ 相对于二维数组来说，这样的组织方式允许不等长的字符串能够被以一种相对规整的方式组织在一起。看上去的效果就好像数组中的每个元素就是一个字符串一样，尽管每个元素只是指向某字符串的指针。
- ☑ 指向字符串的指针数组的初始化更为简单。各个字符串都是可以分别定义的，只要让数组中的指针指向各字符串即可。

下面给出完成的示例代码清单，该程序首先初始化一个指针数组，让其中的每个指针都指向一个字符串，再对这些字符串进行排序（使用的方法是前面已经介绍过的“冒泡法”），最后按顺序输出各字符串。

```
#include "stdio.h"
#include "string.h"

#define SIZE 6

//按字符表排序
void sort(char * name[], int n)
{
    char * tmp;
    int i = n - 1, k;
    for(; i >= 0; i--)
    {
        for(int j = i-1; j >= 0; j--)
        {
            k = i;
            if(strcmp(name[k], name[j]) < 0)
            {
                tmp=name[i];
                name[i]=name[j];
                name[j]=tmp;
                k=j;
            }
        }
    }
}

//输出数组中指针所指向的字符串
void output(char * name[], int n)
{
    for(int i = 0; i < n; i++)
        printf("%s\n", name[i]);
}

void main()
{
    char * name[SIZE] = {"George Washington",
                        "Thomas Jefferson",
                        "Abraham Lincoln",
                        "Franklin Delano Roosevelt",
```



```

        "Ronald Reagan",
        "Barack Obama"};
sort(name, SIZE);
output(name, SIZE);
}

```

上述代码告诉我们，指针数组也可以用作函数参数，这一点请读者留意。完成编码后读者可自行编译并运行上述程序。

通常而言一个数组中每个元素都是同类的，例如整型、字符型等。这就意味着每个元素的大小也是相等的。但是指针数组提供了这样一种可能：就是让数组中的每个元素可以并不完全严格等同。指针数组仅仅要求数组中的每个元素都属于同一指针类型。但是指针是指向数组的，而指针所指向的数组具体有多大并不影响这些同类型指针的同类性。

这样一来，通过数组中所存储的指针元素就能够实现将很多并不严格统一的数组数据组织到一起。这样的实现方式非常灵活。特别在组织一些树形或者网状复杂结构时，都借鉴了这种数据组织思想。在读者日后学习数据结构的时候，不妨再回想一下树和图的链式存储是不是有点跟我们这里所讲到的结构组织方式有异曲同工之妙。

7.5.2 指向指针的指针

如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。打个比方来说，如果现在有一个古董花瓶，以为它很贵重，所以我们把它锁在一个保险柜 A 里。那么这个花瓶就相当于一个变量，而用来打开保险柜 A 的钥匙 Key1 就是一个指针，它指向了花瓶。

但是，可能我们还是有点不太放心，因为我们不想把这把钥匙整天揣在身上，于是我们又把钥匙 Key1 锁进了另外一个保险柜 B 里。这时我们随身带着的就是用来开启保险柜 B 的钥匙 Key2。那么 Key2 就相当于一个指向指针的指针。

1. 间接访问与二级间接访问

在前面已经介绍过，通过指针来访问变量的方式称为“间接访问”。由于指针变量直接指向变量，所以称为“单级间接访问”。而如果通过指向指针的指针变量来访问原始变量则构成了“二级间接访问”，如图 7.22 所示。当然，既然有“二级间接访问”，那么三级、四级……也是可以的，但是实际中很少使用超过二级的间接访问方式。级数越多，关系越乱，越容易出错。

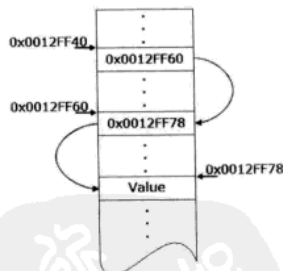


图7.22 二级间接访问

2. 指向指针的指针的实际用途

将一个指针指向另外一个指针，并通过多级间接访问的方式获取实际变量值好像多少有点舍近求远，那么指向指针的指针到底有什么实际用途呢。在前面我们已经向读者介绍了指针数组的使用，如果读者能够在头脑中建立起指针数组的模型，那么就很容易理解为什么要使用指向指针的指针了。

再回忆一下如图 7.21 所示的指针数组的例子，可见 name 是一个指针数组，它的每一个元素都是一个指针型数据。数组名 name 代表该指针数组的首地址，name+1 也就是 name[i]



的地址。既然指针变量可以指向一个数组，那么我们也可以设置一个指针变量 `p`，使它指向指针数组 `name`。这时，`p` 就是指向指针型数据的指针变量。现在读者应该很容易理解为什么要使用指向指针的指针，指向指针的指针可以认为是指针数组的另外一种访问方式。一个指向指针的指针就可以用来表示一个指针数组。

3. 指向指针型数据的指针变量定义

通常，定义一个指向指针型数据的指针变量的基本方法如下：

类型名 ** 指针变量名；

例如，下面这段示例程序使用了指向指针的指针这种语法形式：

```
#include <stdio.h>

void main()
{
    double d = 111.111, *p, **pp;
    pp=&p;
    p=&d;
    printf("d1 = %8.3f, ", **pp);
    **pp+=222.222;
    printf("d2 = %8.3f\n", d);
}
```

编译并运行上述程序，其输出结果如下：

```
d1 = 111.111, d2 = 333.333
```

从上述定义中可见，变量名 `pp` 的前面有两个“*”号，这相当于 `*(**pp)`。显然 `*pp` 是指针变量的定义形式，如果没有最前面的“*”，就表示定义了一个指向 `double` 型数据的指针变量。现在前面又多了一个“*”号，就表示指针变量 `pp` 指向了一个 `double` 型指针变量，而 `*pp` 正是 `pp` 所指向的另一个指针变量。

在本例中，`*pp` 就是指针变量 `p`。如果指针变量 `p` 的地址是 `address_p`，变量 `d` 地址是 `address_d`。此时，`pp` 表示指针变量 `p` 的值（即变量 `d` 的地址 `address_d`），因此表达式 `**pp` 与变量 `d` 等价。

4. “双重指针”访问字符串数组实例

在理解了指向指针的指针语法之后，我们再举一个通过这种“双重指针”来访问字符串数组的例子。读者也可以思考一下，如何使用指针数组来完成这个题目的实现。目前很多（例如银行、邮局等）服务窗口前面都有一个电子评分器，当顾客接受完服务之后，他会按一个数字来表示对业务员所提供服务的满意程度。

现在就请编程模拟这样一个电子评分器，要求使用一个指向指针数组的指针来完成对相应评语的存取操作。这里假定满意程度分为 5 个等级：若输入 0，则屏幕显示“Very Bad”；若输入 1，则屏幕显示“Bad”；若输入 2，则屏幕显示“Moderately Well”；若输入 3，则屏幕显示“Good”；若输入 4，则屏幕显示“Excellent”。下面给出程序实现的代码清单：

```
#include <stdio.h>

void main()
```

```
{
    int grade;
    char *str[] = {"Very Bad", "Bad",
                  "Moderately Well", "Good", "Excellent"};

    char **pp;
    pp = str;
    printf("请输入等级分(0~4): ");
    scanf("%d", &grade);
    printf("%s\n", *(pp+grade));
}
```

简单分析一下上面这个程序，其中 `pp` 指向指针数组 `str` 的第一个元素 `str[0]`，`pp+1` 则指向 `str` 的下一个元素 `str[1]`，`pp+2` 指向 `str[2]`，依此类推。`str` 是一个指针数组，该数组中的每个元素都是指向一个字符数组的指针，因此 `*pp` 就等于字符数组“Very Bad”的首地址，`*(pp+1)` 则是字符串“Bad”的首地址，依此类推。

使用指向指针的指针会使得操作变得更加简洁灵活，但是对于初学者而言，指向指针的指针并不太容易掌握，因为在指针指来指去之后很多人就会被绕晕。任何关于指针的操作一旦使用不当，后果都会极其麻烦，因此我们建议读者在对这种语法确实能够清晰理解的基础之上，再去使用它。

7.5.3 main()函数的参数

本书前面所列举的示例程序中，`main()`函数都是不带参数的。可见，无参的 `main()`函数其定义部分的括号里要么是空的，要么只有一个关键字 `void`。而事实上，`main()`函数也是可以带参数的。C 语言规定 `main()`函数的参数只能有两个，习惯上将这两个参数写为 `argc` 和 `argv`。带参数的 `main()`函数其定义可以写为：

```
类型标识符 main (argc, argv)
{
    //函数体
}
```

C 语言中同时还规定 `main()`函数的第一个形参 `argc` 必须是整型变量，而其第二个形参 `argv` 则必须是指向字符串的指针数组。因此，加上形参说明后，`main()`函数的定义应写为：

```
类型标识符 main (int argc, char *argv[])
{
    //函数体
}
```

在介绍函数的时候我们已经讲过 `main()`函数是不能被其他函数调用的，因此 `main()`函数也不可能程序内部获得参数的实际值。那么，我们又该如何把实参值传递给 `main()`函数的形参？这项工作其实是通过操作系统的命令行来完成的。

7.5.4 main()函数参数应用实例

下面我们就举例说明 `main()`函数参数的使用方法。

第1步 请读者在 Visual C++下新建一个名为 `test` 的 Win32 控制台项目，然后在工程中添加一个名为 `test.c` 的源文件并编码，代码如下：

```
#include "stdio.h"
```



```
#include "string.h"

//子串统计函数
int count(char * str, char * substr)
{
    unsigned int i, j, k, num = 0;
    for(i = 0; i < strlen(str); i++)
    {
        k = 0;
        for(j = i; substr[k] == str[j]; k++, j++)
            if(substr[k+1]=='\0')
            {
                num++;
                break;
            }
    }

    return num;
}

int main(int argc, char *argv[])
{
    if(argc != 3)
    {
        printf("Input Error!!!");
        return 0;
    }

    char * str[2];
    int i = 0;
    while(argc-- > 1)
    {
        str[i] = argv[i+1];
        i++;
    }

    int num = count(str[0], str[1]);
    printf("The substring %s appears %d times.\n", str[1], num);

    return 1;
}
```

第2步 完成编码后请编译并运行该程序，编译器最后将生成一个名为 test.exe 的应用程序。在 Windows XP 系统下执行“开始”→“运行”命令，弹出“运行”对话框，如图 7.23 所示，在“打开”文本框中输入命令“cmd”并单击“确定”按钮，这样控制台界面就被启动起来。

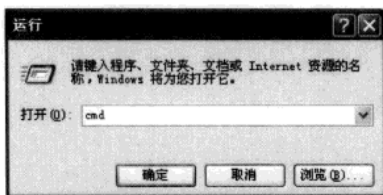


图7.23 “运行”对话框

在控制台中运行一个可执行文件的基本方法是在提示符下键入文件名，再输入实际参数，这样就能够把这实参传送到 `main()` 函数的形参中去。提示符下命令行的一般形式为：

```
C:\>可执行文件名 参数1 参数2...
```

`main()` 函数的形参 `argc` 表示的是接收参数的个数，而第二个形参 `argv` 则是一个指向字符串的指针数组。

`main()` 函数参数需要注意以下几点：

- ☑ `main()` 函数的两个形参和命令行中的参数在位置上不是一一对应的。这是因为 `main()` 函数的形参只有两个，而命令行中的参数个数原则上未加限制。但是通过表示形参个数的 `argc` 与指向形参字符串数组的指针 `argv` 的组合就可以允许 `main()` 函数接受多个参数了，当然这些参数只能是字符串数组类型的值。
- ☑ 形参 `argc` 值不需要我们从控制台给出，它的值是在输入命令行时由系统按实际参数的个数自动赋予的。也就是说系统会自动检测接受到字符串的个数，并将这个数字赋给 `argc`。
- ☑ 还要注意的一个问题是程序的“可执行文件名本身也算一个参数”，因此如果在命令行中输入：

```
C:\>CProgram BASIC FORTRAN JAVA PYTHON
```

那么由于文件名 `CProgram` 本身也算一个参数，所以该程序将共接收到 5 个参数，因此 `argc` 取得的值为 5。

第3步 当控制台界面被启动后，可以在控制台提示符下输入如下的命令，假设该程序位于计算机中的 D 盘根目录下：

```
D:\>test ccccatatcatcatccat cat
```

程序运行结果如图 7.24 所示。

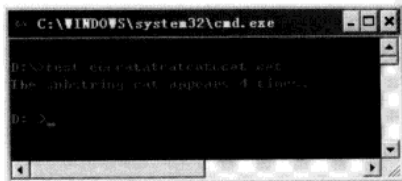


图7.24 程序运行结果演示

示例程序的作用是从命令行接收两个字符串 `str[0]` 和 `str[1]`，然后统计字符串 `str[1]` 在 `str[0]` 中出现的次数。可见，程序正确运行并统计出了子字符串在原字符串中出现的次数为 4。



第 8 章

预 处 理



在前面章节中已多次出现了 `#define` 与 `#include`，含有这些指示的程序段叫做预处理部分。所谓预处理是指源文件在进行编译的第一遍扫描（词法分析和语法分析）之前所做的工作，由预处理器完成。预处理器是一个在编译前对 C 程序进行编辑的工具软件。

预处理为编写大型程序提供了一种良好的模块化的实现方式，使程序的结构性更好。而且也便于程序的阅读、修改、移植和调试。不过，过分依赖预处理器会引发许多不易发现的编程错误，也可能使编写的程序非常难懂。所以，笔者提倡要适度地使用预处理器。

本章从预处理器的工作方式入手，重点介绍预处理器的 3 种主要的形式：宏定义、文件包含和条件编译。



8.1 预处理器概述

由程序员编写的源码文件通常是不会被直接编译的,程序员一般使用预处理器在编译之前对其进行一定的加工,因此了解有关预处理器的工作方式是非常必要的。

程序员与预处理器进行交互的工具是一种被称为预处理器指示的命令,即程序员放在源码文件中的一些以井号“#”开头的单行命令。这些预处理器指示驱使预处理器完成一些任务,而预处理器所执行的任务则依赖于指示本身。本节就对预处理器的工作方式以及预处理器指示进行介绍。

8.1.1 预处理器的生活方式

笔者在第一章中曾经提到过C程序的执行过程分为两步:

第1步 编译生成目标程序。

第2步 连接生成可执行程序。

不过,有一点前面没有交代清楚,那就是C语言编译器编译的程序并不是C语言源程序,而是经过预处理器处理后的程序,为了表述方便,将其称为预编译程序。一个C语言源程序的执行过程如图8.1所示。

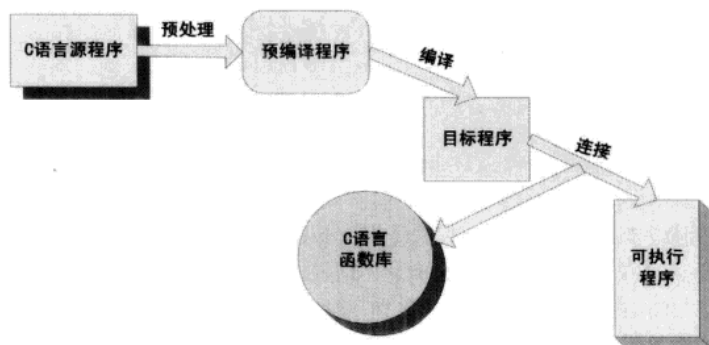


图8.1 C程序执行过程

可以看出,预处理是在编译前对程序进行处理的,输入的是一个C语言源程序,此源程序可能包含预处理指示(如前面所讲的`#include "stdio.h"`)。预处理器会执行这些指示,并在处理过程中删除这些指示。预处理器输出预编译程序,是源程序的一个编辑后的版本,其中不再包含任何指示。而预编译程序成为编译器的输入,编译器检查预编译程序是否有错误,然后编译生成目标程序。也就是说,C程序的执行过程分为3步:

第1步 预处理生成预编译程序。

第2步 编译生成目标程序。

第3步 连接生成可执行程序。




预编译程序和源程序到底有什么不同？预处理器到底对源程序做了些什么？下面通过比较求圆形周长和面积的 C 语言源程序以及其经过预处理器处理后生成的预编译程序来进行说明。求圆形周长和面积的源程序清单如下所示：

```
/*此程序用于计算圆形周长和面积*/
#include <stdio.h>
#define PI 3.1415926
void main()
{
    int r;
    float girth,area;
    girth=2*PI*r;
    area=PI*r*r;
    printf("圆形周长为: %f, 圆形面积为: %f\n", girth,area);
}
```


由预处理器处理后生成的预编译程序为：

```
空行
空行
由 stdio.h 引入的行
空行
空行
void main()
{
    int r;
    float girth,area;
    girth=2*3.1415926*r;
    area=3.1415926*r*r;
    printf("圆形周长为: %f, 圆形面积为: %f\n",grith,area);
}
```

预处理器删除了 `#include` 指示，并通过将 `stdio.h` 的内容加入到程序中来对其进行响应。上面的程序只是为了简单说明，所以没有将 `stdio.h` 的内容完全显示出来。预处理器也删除了 `#define` 指示，并将该文件中出现的所有 `PI` 替换为指定值，此处为 3.1415926。

 **注意：**预处理器并没有删除包含指示的行，而是简单地将其替换为空行。

预处理器能够执行预处理指示，并完成一些替换或引入的操作，但其功能不仅仅如此。像上面的程序中，预处理器还将每一处注释都替换成一个空格字符。有些预处理器还会删除一些不必要的空白字符，诸如语句中用于缩进的空格符和制表符。

 **注意：**预处理器只包含很少量的 C 语法规则，在执行指示时可能会产生一些程序错误，而这些错误又非常隐晦，不易被发现。因此，检查预处理输出的预编译程序是调试程序，特别对于大型程序，这是排错的有效途径之一。

8.1.2 使用 Microsoft Visual C++ 6.0 生成预编译程序

既然预编译程序如此重要，强大的 Microsoft Visual C++ 6.0 当然不会“坐视不管”，使用一些选项就可以生成预编译文件（一般是后缀名为 `.i` 的文件）或者将其在标准输出中展示。

1. 预处理选项

在 Microsoft Visual C++ 6.0 中的各种预处理选项如表 8.1 所示。

表8.1 预处理选项

选 项	输出中是否包含#line 指示	输 出 位 置
/C	否	.obj 文件
/E	是	标准输出 (stdout)
/P	是	.i 文件
/EP	否	标准输出 (stdout)
/E /EP	否	标准输出 (stdout)
/P /EP	否	.i 文件

下面详细介绍其中常用的选项。

☐ /C 选项：此选项处理预处理后的文件，编译或汇编源文件，但是不进行连接。编译器输出对应于源文件的目标文件。Microsoft Visual C++ 6.0 生成的是后缀名为 .obj 的文件。此选项为 Microsoft Visual C++ 6.0 默认选项。

☐ /E 选项：此选项处理 C 和 C++ 源文件。编译器生成对应于源文件的预编译程序。此程序中所有的预处理指示都被执行，所有的宏被展开，注释被删除。结果只会输出在 Microsoft Visual C++ 6.0 的标准输出中，而不生成文件。



提示：如果你不想注释被删掉，可以同时使用 /C 选项与 /E 选项。

☐ /P 选项：此选项与 /E 类似，只是输出为一个 .i 形式的文件，也就是预编译文件。

其他的选项此处不进行介绍，请读者参考 Microsoft Visual C++ 6.0 的帮助文档。



注意：表 8-1 中提到的 #line 指示会在后面进行介绍。此处只需要知道包含此指示的程序在经过预处理后还是保持源文件中的行号。这样可以方便调试，假设源文件中的第 10 行出错，编译器会报出第 10 行出现某种错误。如果不包含 #line 指示，则编译器报的错误出现的行号可能和源文件不符，从而使排错工作变得困难。

2. 一个生成.i 文件的例子

下面展示如何使用 Microsoft Visual C++ 6.0 生成前面所述的求圆形周长和面积程序的预编译文件。

第 1 步 编辑好源文件。

第 2 步 选择菜单 Project (工程) → Settings (设置) 命令，在弹出的对话框中选择 C/C++ 选项卡，在 Project Options 文本框中增加选项 /P，如图 8.2 所示。



图8.2 设置参数

第3步 编译程序。

编译完程序后，即可在源文件所处的目录中生成一个以.i 为后缀名的文件。此文件就是求圆形周长和面积程序的预编译文件，可以用它查看预处理后的结果，帮助调试程序。

8.1.3 预处理指示分类

前面见到的#include、#define 都属于预处理指示，同时也是预处理器要处理的主要对象。C 语言中的预处理指示分为以下几种类型。

- ☑ 宏定义：包括定义宏的指示#define 和删除宏的指示#undef。
- ☑ 文件包含：使用#include 指示使一个指定文件的内容被包含到程序中。
- ☑ 条件编译：使用#if、#ifdef、#ifndef、#elif、#else 和#endif 指示，可以根据编译器测试的条件来对一段文本块选择性地编译或者不编译。
- ☑ 其他预处理指示：包括#error、#line 和#pragma 指示。
- ☑ 另外两个运算符：“#”和“##”，这两个运算符在预处理时被执行，而不是在编译时被执行。

8.1.4 预处理指示规则

俗话说：“无规矩不成方圆”，C 语言中的预处理指示也必须遵守一些规则才能使预处理器对其进行很好的处理。以下是 C 语言中所有预处理指示都必须遵守的规则。

- ☑ 预处理指示都以“#”开始。需要注意的是，“#”号不一定在一行的行首，如果不在行首，预处理器要求“#”号之前必须只包含空白字符。例如，下面是合法的预处理指示：

```
#define PI 3.1415926
```

- ☑ 在指示的符号之间可以插入任意数量的空格或横向制表符。例如，下面是合法的预处理指示：

```
#   define      PI   3.1415926
```

- ☑ 如果不明确指明预处理指示，指示总是在第一个换行符处结束。如果想在下一行继续指示，必须在当前行的末尾使用“\”字符。例如，下面的预处理指示定义了输出 5 个 int 型数：

```
#define PF(one,two,three,four,five) printf("%d,%d,%d,%d,%d\n",\
one,two,three,four,five)
```

☑ 指示可以出现在程序中的任何地方。

通常程序员习惯将#define 和#include 指示放在程序的开始,其他指示则放在后面。但这不是必需的,指示在程序中的位置完全不受限制,它甚至可以出现在函数定义中。

☑ 注释可以与指示放在同一行中。

实际上,在一个预处理指示后加一个注释来对指示本身进行解释是一种非常好的编程习惯。下面的指示就遵循了这种习惯:

```
#define PI 3.1415926 //定义表示圆周率的宏
```

8.2 宏定义

对于宏定义(简称宏),读者并不陌生。在第2章介绍符号常量时,其实已经“认识”它了。不过,这个“老朋友”并不只是单单定义一下符号常量那么简单,宏定义其实分为带参数和不带参数两种。本节就来详细介绍一下C语言中的宏定义。

8.2.1 无参宏定义

1. 无参数的宏定义形式

不带参数的宏定义的一般形式如下:

```
#define 标识符 替换文本
```

其中,替换文本是一种C语言记号,包括标识符、关键字、数、字符常量、字符串常量、运算符和标点符号。当预处理器遇到一个宏定义时,会用“替换文本”来代替程序中任何位置出现的“标识符”。此过程叫做宏替换。标识符又被称为宏名。

2. 无参数的宏定义作用

☑ 无参宏定义经常被用于给数值、字符和字符串命名。

无参宏定义主要被用于定义那些被 Kernighan 和 Ritchie 称为“明示常量”(manifest constant)的东西。例如:

```
#define ARRAY_LEN 80 //指定数组长度为80
#define TRUE 1 //指定真的值为1
#define PI 3.14159 //指定PI的值为3.14159
#define CR '\r' //指定CR表示换行符
#define END '\0' //指定结束为\0
```

上述的宏定义指示都会用替换文本部分代替标识符部分,即源程序中所有出现标识符的地方在预编译程序中将不再出现,在相应的位置出现替换文本。

☑ 对类型重命名。

例如,下面的指示完成将 long double 类型重命名为 LD:

```
#def
```



不过,这并不是定义新类型的最佳选择,定义新类型的最好方法是下一章要介绍的用户自定义类型。

☑ 对 C 语言语法进行细微修改。

“修改 C 语言语法”,这听起来似乎有些“大逆不道”。这样做其实是为了熟悉其他语言(如 Pascal)的人能够更容易地掌握 C 语言。例如,下面的宏定义为 C 语言中的两个符号“{”和“}”重命名,使其更像 Pascal 语言的语法:

```
#define BEGIN {  
#define END }
```

甚至可以通过这种方式创造出自己的语言。例如,下面发明出一个 FOREVER “语句”,代表一个无限循环:

```
#define FOREVER for (;;) 
```

不过,笔者并不认为这是一个好的编程习惯,因为程序员写的程序往往是要让别人看懂的,而这样的程序易读性差,很难被理解。

3. 无参数的宏定义优点

现在来想想 C 语言中为什么要引入宏定义呢?笔者认为出于宏定义的以下优点。

☑ 简化输入并防止输入出错

宏定义使用一个标识符来替换源文件中的一个替换文本,这可以有效地简化程序中重复出现的某些复杂替换文本的输入工作。从求圆形周长和面积的程序很容易发现仅使用了两个字符“PI”就替换了原来很长的一段数字,当程序中该数字频繁出现时,不难想象使用宏是多么的方便。

宏定义还可以防止一些输入错误。假想常量 3.14159 在程序中多次出现,在没有使用宏定义的情况下,谁能保证每次都输入正确,将其输入成 3.14195 或 3.1416 的可能是常有的事。

☑ 提高程序的可读性

对于一串数字或者字符往往可能有很多种解释,但是在某些程序中将其定义成常量来使用往往是有特定含义的,这就形成了所谓的“魔法数”。使用宏定义来替换这些常量,可以很容易地从宏名中看出它所代表的特定含义,而无须过多关心具体数值。当然,对于宏名,必须下些功夫,务必做到“见名知意”。

☑ 方便程序的维护

假想一下最初你定义的 PI 的数值为 3.14159,但是现在你希望程序能够进行更高精度的计算,所以希望将 PI 的替换文本改为 3.1415926。如果没有使用宏定义,那么对程序中出现替换文本进行逐个修改不但工作量繁重而且极易出错。但是如果使用了宏定义,那么此修改工作就变得方便且高效,因为你只需要修改一下定义本身即可。

8.2.2 带参宏定义

1. 带参数的宏定义形式

C 预处理器支持带参数的宏定义。其一般形式如下:

```
#define 标识符(参数表) 替换文本
```

其中，参数表中可以有一到多个参数，也可以表示成如下形式：

```
#define 标识符(x1,x2,...,xn) 替换文本
```

这些参数根据需要可以在替换文本中出现若干次。其运行规则是：当预处理器碰到一个带参数的宏，会将定义存储起来以便后面的程序使用。在后面的程序中，任何位置如果出现了如下形式：

```
标识符(y1,y2,...,yn)
```

其中，y₁, y₂, ..., y_n是一系列标记，预处理器会使用替换文本对其进行替代，并使用y₁替换x₁, y₂替换x₂, 依此类推。来看下面这个例子：

```
#include <stdio.h>

#define MAX(x,y) ((x)>(y)?(x):(y))

void main()
{
    //定义变量
    int i = 0;
    int j = 1;

    printf("两个数中的最大数是: %d\n",MAX(i,j));
}
```

此程序中的输出语句在替换后变为：

```
printf("两个数中的最大数是: %d\n", ((i)>(j))? (i):(j));
```

2. 带参数的宏定义作用

带宏定义经常被用作模板，来处理程序中要经常重复书写的代码段。例如，你是否已经厌烦了在程序中书写下面的代码段？

```
printf("%d\n", i);
```

其实这条语句的作用就是以%d的形式显示整数。完全可以定义下面的宏，使显示工作从形式上看简单而且清晰：

```
#define PF_INT(i) printf("%d\n",i)
```

有了这个宏定义，就可以在程序中写出下面的输出语句：

```
PF_INT(x*y/z);
```

预处理器会将其转换为：

```
printf("%d\n", x*y/z);
```


8.2.3 带参宏定义与函数

带参宏定义看上去有点像普通的C函数。实际上，带参数的宏也经常用来作为一些简单的函数使用。比如下面的例子：

```
#define getchar() getc(stdin)
```

你也许会问：这不就是“stdio.h”中的getchar()函数吗？不错，确实是，但getchar确实是个宏定义，而不是函数。前面为了表述方便，将其作为一个函数来介绍。



 **注意：**带宏定义参数表也可以为空，像上面的 `getchar()` 就是。

虽然带宏定义和函数极为相似，但是二者是有区别的，主要表现在以下几个方面：

- ☐ 函数调用是在程序运行时处理的，必须为形参分配临时的内存单元。而宏的展开则是在编译前进行的，在展开时并不分配内存单元，也不进行值传递，只是进行简单的替换。也就是说，宏替换不占运行时间，而函数调用则占用运行时间。这也就是宏替换一般会比函数调用快的原因所在。
- ☐ 宏名没有类型，其参数也没有类型。而函数的参数和返回值是有特定类型的。也就是说，宏更“通用”一些。只要预编译程序是合法的，宏替换可以接受任何类型的参数。例如，如果使用前面所定义的 `MAX` 宏选择两个数中的最大数，参数的类型不仅可以是程序中的 `int` 型，也可以是 `short`、`long`、`float`、`double` 等数据类型。
- ☐ 函数调用只有一个返回值，而宏替换则可以设法得到多个结果。

例如，此处改写前面求最大值的宏定义，使其得到最大值的同时得到最小值。改写后的程序清单如下：

```
#include <stdio.h>

#define MAX_MIN(x,y,max,min)  max=((x)>(y))?(x):(y);  min=((x)<(y))?(x):(y);

void main()
{
    int i = 3;
    int j = 4;
    int max = 0;
    int min = 0;

    MAX_MIN(i, j, max, min);

    printf("两个数中的最大数是: %d\n",max);
    printf("两个数中的最小数是: %d\n",min);
}
```

- ☐ 函数调用不会使源程序变长。而多次使用宏定义时，因为每处宏替换都会插入宏的替换文本，所以宏替换后的预编译程序会很长。特别是宏嵌套调用时，此问题更加突出。例如，下面的语句完成求 3 个数中的最大数：

```
max = MAX(x, MAX(y, z));
```

预处理后变为：

```
max = ((x)>(((y)>(z))?(y):(z)))?(x):(((y)>(z))?(y):(z)));
```

- ☐ 宏参数没有类型检查，而函数调用有。

当一个函数被调用时，编译器会检查每一个参数来确认其是否是正确的类型。如果不是，或者将参数转换为正确的类型，或者由编译器产生一个出错信息。预处理器则不会检查宏参数的类型，也不会进行类型转换。这可能会引起一些难以发现的程序错误。

- ☐ 无法用一个指针来指向一个宏。但可以用一个指针指向一个函数。

C语言允许指针指向函数，这在第7章“指针”一章中已经进行了介绍。而宏定义会在预处理过程中被删除，所以也不存在“指向宏的指针”这类东西。

8.2.4 使用宏时注意事项

在使用宏时应注意以下几点：

☐ 宏定义中的各种额外符号会被视为待替换文本，可能会引起一些编程错误。

宏定义不是C语句，因此其行末不需要添加分号。如果添加了分号，那么分号也会一同视为待替换的文本。例如：

```
#define LENGTH 100; //加入了分号
int a[LENGTH];
```

上述语句在宏替换后变为：

```
int a[100;];
```

这条语句明显是不能编译通过的。除分号外，最常见的是在指示中加入了等号。例如：

```
#define LENGTH = 100; //加入了等号
int a[LENGTH];
```

上述语句也是无法编译通过的，因为经过宏替换后变为：

```
int a[=100];
```

一般情况下，编译器会将每一处使用到此宏的地方标为错误，但不会直接找到错误的根源——宏定义本身，因为宏定义已经被预处理器删除了。

☐ #define 指示通常出现在程序的开头部分，但它也可以出现在程序的任何地方。宏替换的有效范围是从其定义命令之后到此源文件结束。但是可以使用#undef 指示来终止宏定义的作用域。


#undef 指示的一般形式为：

```
#undef 标识符
```

其中，标识符是之前定义过的一个宏名。例如，下面的指示取消了前面定义的代表圆周率的宏：

```
#undef PI
```

#undef 指示的一个重要作用是取消一个宏的现有定义，以便于重新定义。

 **注意：**如果#undef 后面的标识符没有被定义为一个宏，则不会起任何作用。

☐ 宏定义可以嵌套使用。例如：

```
#define R 1.5
#define PI 3.14159
#define L 2*PI*R
#define S PI*R*R
```

预处理器会不断重新检查替换文本，直到将所有的宏名字都替换掉为止。例如下面的语句：

```
printf("%f\n",L);
```

预处理器在遇到语句中的L时，首先将其替换为(2*PI*R)，接着预处理器重新检查替



换文本，发现 PI 和 R 也是宏，所以需要再次替换为 $2*1.5*3.14159$ 。

☐ 程序中双引号内的内容不会发生宏替换。

例如，下面的语句输出圆周率 π （宏名为 PI）：

```
printf("PI:%f\n",PI);
```

其中第一个 PI 因为包含在引号内，并不会被替换掉，而后面的 PI 则要进行宏替换。

☐ 宏替换与定义变量不同，宏替换只进行字符替换，此替换是在编译前完成，并不分配内存空间。而定义变量是在编译时才被执行，必须为其分配内存空间。

☐ 宏名是一个完整的记号，预处理器将其作为一个整体来替换，而不替换包含宏名的标识符、字符串常量和字符串变量名。

来看下面的程序段：

```
#define LENGTH 100
int ARRAY_LENGTH=200;
if(ARRAY_LENGTH> LENGTH)
    printf("ARRAY_LENGTH 没有被宏替换");
```

预处理器只会将 if 语句条件中的 LENGTH 替换为 100，而不会替换 ARRAY_LENGTH，所以程序将执行 if 语句中的输出。

☐ 宏不能被重复定义，除非新的定义与旧的定义是一样的。

预处理器允许一些小的间隔上的差异，但是宏的替换文本中的记号必须一致。如果含有参数，参数也必须一致。可以使用后面要介绍的条件编译来防止宏被重复定义。

8.2.5 至关重要的圆括号

细心的读者可能已经发现，在本章前面定义的宏的替换文本中有大量的圆括号。是笔者书写代码时的啰嗦表现还是确实有此种需要？先来看下面的宏定义：

```
#define MUL(x,y)    x*y
```

可以看出，此宏是用来计算两个参数相乘后的结果的，而如果程序中包含下面语句：

```
result=MUL(5+3,6);
```

预处理器对其进行宏替换后变为：

```
result= 5+3*6
```

很明显，实际的结果违背了起初的预想。要得到预想的结果，可使用下面的宏：

```
#define MUL(x,y)    ((x)*(y))
```

所以，在宏的替换文本中使用圆括号是非常有必要的。因为省略圆括号会产生意料之外、非预期的结果。

知道了使用圆括号的必要性后，如何正确使用成为一个问题。对于在一个宏定义中哪些地方要加圆括号有如下两条规则。

☐ 如果宏的替换文本中有运算符，那么始终要将替换文本放在括号中。例如：

```
#define TWO_PI (2*3.14159)
```

原因是：圆括号的运算符级别最高，此做法可以使宏替换按照编程者的思路进行，而不会因为运算符的优先级和结合性出现非预期的结果。

☐ 如果是带参宏定义，每次参数在替换列表中出现时都要放在圆括号中。例如：

```
#define PER(x) ((x)/100)
```

需要提醒的是，仅给替换文本添加圆括号是不够的，参数的每一次出现都要添加圆括号。例如，假设 PER 定义如下：

```
#define PER(x) (x/100)
```


如果程序中包含下面语句：

```
j = PER (i+1);
```

预处理器对其进行宏替换后变为：

```
j = (i+1/100);
```

这明显不是预期结果，而“罪魁祸首”就在于没有给参数加圆括号。

 **注意：**在宏定义中缺少圆括号会引起一些隐晦的错误，但程序通常仍然能编译通过，而且宏大多也可以工作，仅在少数情况下会出错。所以一定要合理地使用圆括号。

8.2.6 预定义宏

C 语言提供给用户宏定义的方式完成许多操作，自己当然也“留了一手”，预定义了一些有用的宏。预定义宏主要是提供当前编译的信息。C 语言中的预定义宏如表 8.2 所示。

表8.2 预定义宏

宏 名	描 述
<code>__LINE__</code>	被编译的文件的行数
<code>__FILE__</code>	被编译的文件的名字
<code>__DATE__</code>	编译的日期（格式“Mmm dd yyyy”）
<code>__TIME__</code>	编译的时间（格式“hh:mm:ss”）
<code>__STDC__</code>	如果编译器接受标准 C，那么值为 1

其中，宏 `__LINE__` 和 `__STDC__` 是整型常量，其他 3 个宏是字符串常量。

1. 控制版本的宏

预定义宏 `__DATE__` 和 `__TIME__` 通常用来在编译后的程序中加入一个时间标志，以区分程序的不同版本。所谓版本，是指用于描述同一事物的相互之间有差异的各种形式、状态或内容。作为软件产品的版本，可以简单地理解为每次修改后得到的程序或者软件产品。例如，下面的函数就可用于检查程序版本：

```
void print_version_info(void)
{
    printf("编译日期是: %s\n", __DATE__);
    printf("编译时间是: %s\n", __TIME__);
}
```

其中，`__DATE__` 宏含有形式为月日年的字符串，表示源文件被编译时的日期。源程序编译到目标程序的具体时间作为字符串包含在 `__TIME__` 宏中，形式为时:分:秒。



2. 发现错误的宏

在调试程序时，可以使用 `__LINE__` 宏和 `__FILE__` 宏来找出错误。例如，在程序中经常要做除法运算的被除数是否为 0 的判断，而当程序中包含多条除法运算时，编译器只会终止程序，而不会提示到底是哪条除法运算出错。不过下面的宏却能为我们排忧解难：

```
#define CHECK_ZERO(divisor)\
if (divisor == 0) \
printf("***为 0 的被除数出现在%s 文件的第%d 行***\n", __FILE__, __LINE__)
```

此宏在程序编译前被调用，假设名为 `division.c` 的程序中有如下语句：

```
008 CHECK_ZERO(y);
009 z = x/y;
```

如果 `y` 的值是 0，则会出现如下形式的信息：

```
***为 0 的被除数出现在 division.c 文件的第 8 行***
```

类似这样的错误检测的宏还有很多，而且非常有用。此处不再介绍，请读者查看相关文档。

对于 `__STDC__`，笔者会在“条件编译”一节中介绍。

8.3 条件编译

在 C 语言中，`#define` 指示除了能够定义宏外，还有一个重要的功能就是支持条件编译。前面所示的所有源文件中的所有内容都需要进行编译。不过有时可能希望源文件中某些部分在满足特定条件的情况下才进行编译，这就是所谓的“条件编译”。前面之所以没有提到它，是因为之前的程序都是一些小规模教学用例。而当程序的规模变大后，条件编译的作用和优势就会突显出来，实际开发中的程序也往往要使用条件编译，因此掌握它是非常有必要的。

8.3.1 条件编译的形式

条件编译分为以下 3 种形式。

1. 第一种形式

```
#ifdef 标识符
    程序片段 1
#else
    程序片段 2
#endif
```

上述指示的意义是：如果标识符已经被 `#define` 指示定义过，则对程序片段 1 进行编译；否则对程序片段 2 进行编译。

此形式的流程图如图 8.3 所示。



图8.3 第一种形式条件编译流程图

其中的程序片段 2 可以为空, 这时 `#else` 部分就没有任何意义了, 可以将其省略。下面的程序清单展示了这种条件编译的用法:

```
#include <stdio.h>

#define DEBUG

void main()
{
    #ifdef DEBUG
        printf("Begin...\n");
    #else
        printf("Cannot Begin!\n");
    #endif

    int array[10], i;
    for (i = 0; i < 10; i++)
    {
        array[i] = i;

        #ifdef DEBUG
            printf("i = %d\n", i);
            printf("array[i] = %d\n", array[i]);
        #endif
    }
}
```

指示中的 `#ifdef` 也可以写作 `#if defined`。预处理器指示 `#if` 类似于 `if` 语句, 其差别在于 `#if` 必须与相应的一个 `#endif` 配对。在上面的程序中, 预处理器判断 `DEBUG` 是否被指示 `#defined` 定义。因为程序的第 3 行的语句 `#defined DEBUG` 已对其进行了宏定义, 因此 `#ifdef` 中的程序片段会被编译, 而 `#else` 中的就不会。

可以看出, 指示 `#ifdef` 和指示 `#else` 将程序分成不同的片段并将其隔离开。此特点在程序调试时特别有用。程序员只需将某些调试代码写进 `#ifdef` 或 `#else` 中的程序片段里, 只有在满足调试的条件时, 调试代码才会被编译, 而不会影响到程序的正常运行。

2. 第二种形式

```
#ifndef 标识符
    程序片段 1
#else
    程序片段 2
#endif
```

其中, `#ifndef` 等价于 `#if !defined`。

上述指示的意义是: 如果标识符未被 `#define` 命令定义过, 则对程序片段 1 进行编译, 否则对程序片段 2 进行编译。

此形式与第一种形式的功能正好相反, 区别为: `ifdef` 关键字换成了 `ifndef` 关键字。

此形式的流程图如图 8.4 所示。

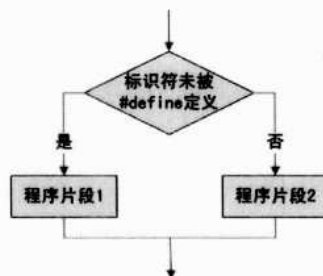


图8.4 第二种条件编译流程图

3. 第三种形式

```
#if 常量表达式  
    程序片段 1  
#else  
    程序片段 2  
#endif
```

上述指示的意义是：如果常量表达式的值为真（非 0），则编译程序片段 1，否则编译程序片段 2。

此形式的流程图如图 8.5 所示。

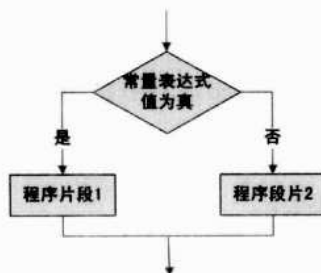


图8.5 第三种条件编译流程图

此形式看起来更像前面介绍的 if 语句。对应于 if 语句的 else if 形式，此形式还有一种形式可以实现多重选择条件编译，也可以称为嵌套形式的条件编译，其形式如下所示：

```
#if 常量表达式 1  
    程序片段 1  
#elif 常量表达式 2  
    程序片段 2  
#else  
    程序片段 3  
#endif
```

上述指示的意义是：如果常量表达式 1 为真，则编译程序片段 1；否则，如果常量表达式 2 为真，则编译程序片段 2；否则，就编译程序片段 3。

此形式的流程图如图 8.6 所示。

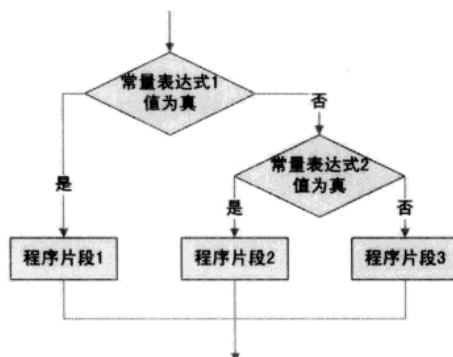


图8.6 嵌套条件编译流程图

8.3.2 条件编译的作用

☑ 编写在多台机器或多种操作系统之间可移植的程序。

下面这个“神奇”的程序中会根据 Windows、DOS 或 UNIX 是否被定义为宏，而将 3 组代码之一包含到程序中：

```
#if defined(Windows)
...
#elif defined(DOS)
...
#else defined(UNIX)
...
#endif
```

在程序的开始处会设定这些宏中的一个（且仅有一个），则会由此选择一个特定的操作系统。假设定义为 UNIX，则程序将运行在 UNIX 操作系统上。

☑ 编写在不同的编译器上进行编译的程序。

不同的编译器经常使用不同的 C 语言版本，这些版本之间会有一些差异。其中有些会接受标准 C，另外一些则不会。一些版本会包含针对特定机器的扩展，而其他的可能没有，或提供不同的扩展集。使用条件编译可以使程序适应于不同的编译器。来看下面的程序：

```
#if __STDC__
    标准 C 函数原型
#else
    其他 C 函数声明
#endif
```

__STDC__ 是用于判断编译器是否支持标准 C 的宏。如果值为真，则表示支持标准 C，运行标准 C 的函数；如果不支持，则执行编译器支持的其他 C 函数的声明。对于每一处函数的声明，都可以使用上面的代码。

☑ 防止重复包含。关于条件编译如何防止重复包含将在下面一节中进行介绍。



8.4 文件包含

文件包含是 C 语言中允许源码文件使用共享代码的一种特性。假设某个源文件中需要使用到的变量或函数是在其他文件中被声明的，那么在 C 语言中，就必须通过将该变量或函数的声明包含到此源程序中来对其进行访问。所谓“文件包含”就是指一个源文件可以包含另外一个源文件的全部或部分内容。C 预处理器通过文件包含机制来自动地完成对变量或者函数声明添加，而不用手动地去增加。

8.4.1 头文件

“头文件”这个词并不陌生，例如前面的程序中多次出现的“stdio.h”就是标准库的头文件。但到目前为止，似乎还有些“犹抱琵琶半遮面”的感觉，现在笔者就来揭开这层面纱，详细介绍头文件。

1. 头文件和定义文件

在通常情况下，每个 C 程序由头文件（header files）和定义文件（definition files）组成。头文件作为一种包含功能函数、数据接口声明的载体文件，用于保存程序的声明，而定义文件用于保存程序的实现。

之所以将程序分成头文件和定义文件两部分，是因为当程序的规模变大时，程序设计就不是一个人能完成的工作了。当人们相互合作去完成这个工作时，势必会将程序模块化，每个人完成自己的模块，而这种模块化的具体实现就是划分为不同的文件。这种方式又引起了一个需要考虑的问题，即这些文件之间的定义与说明的共享问题。

要尽可能使所要共享的部分集中在一起，当要对程序进行改进时也能保证程序的正确性。而这些共享的部分在 C 语言中的具体实现就是将其放入头文件中。每个程序员在开发分配给自己的模块时，如果需要使用这些头文件中的内容，就可以使用#include 进行引用。

2. 典型的头文件

C 语言中的头文件是以.h 为后缀名的，下面展示一个典型的头文件：

```
/**
 *xxx.h - ...
 *
 *      Copyright (c) ...
 *Purpose:
 *      ...
 */
#ifndef DEF_H          //防止 def.h 被重复引用
#define DEF_H

...
#include <stdio.h>      //引用标准库的头文件
#include "mylib.h"      //引用自定义的头文件
...
void function1(...);   //函数声明
...
#endif
```

可以看出，头文件一般由3部分组成：

- ☐ 头文件开头处的版权和版本声明。其一般用注释的方式给出。
- ☐ 预处理块。用条件编译和宏定义实现。
- ☐ 函数和结构的声明等。

从上面的示例中还可以看出，头文件的主要作用是调用库功能，对每个被调用函数给出一个描述，其本身不包含程序的逻辑实现代码，只起描述性作用，告诉程序通过相应途径寻找相应功能函数的真正逻辑实现代码。程序只需要按照头文件中的接口声明来调用库功能，编译器就会从库中提取相应的代码。

可以说，头文件是程序和函数库之间的桥梁和纽带。在整个软件中，头文件并不是最重要的部分，但也是不可缺少的部分。犹如一本书中的目录，读者（用户程序）可以通过目录很方便查阅需要的内容（函数库），但目录毕竟不是书的核心。

8.4.2 文件包含的形式

文件包含的一般形式如下：

```
#include <文件名>
```

或：

```
#include "文件名"
```

以上两种方式（使用双引号和使用尖括号）都可以实现文件包含，二者的区别在于：

- ☐ 使用尖括号时，预处理器到存放库函数头文件所在的目录中寻找要包含的文件，而这些文件大多定义在标准库中，这种方式被称为标准方式。
- ☐ 使用双引号时，系统先在用户当前的目录中寻找要包含的文件，此目录中存放的可以认为是用户自定义函数库。如果找不到，则再按标准方式进行查找。

例如下面的两条语句：

```
#include <include_file.h>
#include "include_file.h"
```

其查找过程如图 8.7 所示。

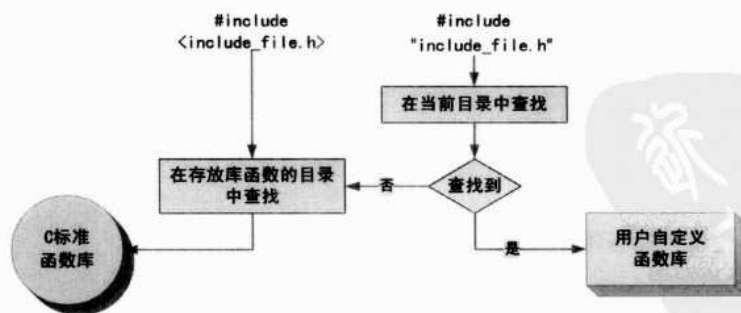


图8.7 两种文件包含查找过程

可见，如果为了调用标准库函数而包含文件，那么应该使用尖括号形式，这样可以节省查找时间。相反，如果包含的文件是由用户自己编写的，那么这时最好使用双引号形式。



这是一个好的编程习惯，可以使程序员能够区别被引入的文件是自己开发的还是来自于标准库的。

可见，程序员通过`#include` 指示来命令预处理器执行文件包含。`#include` 指示将文件名或者库名作为一种参数来对待。每碰到一个`#include` 指示，预处理就自动地用指定的文件内容来替换该指示行。来看下面这个程序片段：

```
#include <stdio.h>
#include <math.h>

#include "my_lib.h"
#include "my_function.h"

#include "..\file1.h"
#include "directory\file2.h"

int main()
{
    //...
}
```

上述程序示例告诉读者使用双引号时不仅可以指定文件名，还可以指定文件的全路径名，这时，预处理器会从指定路径开始查找，如果查找不到，则再按标准方式进行查找。

8.4.3 使用文件包含时注意事项

在使用文件包含时应注意以下几点：

- ☑ 一个 `include` 命令只能指定一个被包含文件，若有多个被包含文件，则需用多个 `include` 命令。

例如，下面的指示是非法的：

```
#include <file1.h> <file2.h>
```

正确的形式为：

```
#include <file1.h>
#include <file2.h>
```

- ☑ 文件包含允许嵌套，即在一个被包含的文件中又可以包含另一个文件。预处理器会将所有被包含的文件层层剖析，即不管是被包含文件的内容，还是嵌套被包含文件的内容，都可以在程序中引用。
- ☑ 使用文件包含时，特别是嵌套包含时，要特别注意防止文件被重复包含。可以使用条件编译来防止重复包含。

事实上，使用条件编译防止重复包含非常常用，例如：

```
#ifndef FIRSTINCLUDE_H
#define FIRSTINCLUDE_H
...
#endif
```

指示`#ifndef` 检查 `FIRSTINCLUDE_H` 之前是否已经被定义，此处的 `FIRSTINCLUDE_H`

是一个预编译器常量，一般被写成大写字母，如果 `FIRSTINCLUDE_H` 在前面没有被定义则从 `#ifndef` 到 `#endif` 之间的所有语句都被包含进来进行处理；否则 `#ifndef` 与 `#endif` 之间的行都将被忽略。

为了保证头文件只被包含一次，需要将如下 `#define` 指示：

```
#define FIRSTINCLUDE_H
```

放在 `#ifndef` 后面，这样在头文件的内容第一次被处理时 `FIRSTINCLUDE_H` 将被定义，从而防止了被再次定义。

☐ 包含文件和被包含文件在编译后会成为一个文件，而不是两个或者多个文件。所以不管是出现在包含文件，还是被包含文件中的变量和函数都应视为在一个文件中来处理。

8.5 其他指示

本节将简要地介绍一下 `#error`、`#line` 和 `#pragma` 指示。这些预处理指示并没有前面介绍的指示用途那么广泛，所以 C 语言的初学者可能很少用到。如果你是个“菜鸟”，完全可以跳过本节。但如果你正在向高手迈进，不妨看看这些指示。

8.5.1 #error 指示

`#error` 指示的一般形式为：

```
#error 消息
```

其中，消息是一个 C 语言标记。当预处理器遇到一个 `#error` 指示时，就会显示一个出错信息，此信息一定会包含消息。但对于不同的编译器，出错信息的具体形式也不尽相同。下面是一个典型的出错信息示例：

```
Error directive: 消息
```

一般情况下，出现 `#error` 指示预示着一个严重的程序错误，在此错误导致系统崩溃前给出一个消息。

`#error` 指示一般与条件编译指示一起用于检测正常编译过程中不应出现的情况。

例如，假设某个程序是运行在 32 位机上的，其 `int` 型运行的最大值为 2147483647，则下面的程序对此限制给予检测（其中 `INT_MAX` 为运行最大数的宏）：

```
#if INT_MAX<2147483647
    #error int 类型值太小
#endif
```

假定目前操作的机器为 16 位机，其 `int` 型最大数为 32767。编译器会给出下列出错信息：

```
Error directive: int 类型值太小
```

`#error` 指示经常用于 `#if-#elif-#else` 形式的条件编译的 `#else` 部分，表示所有条件都没有被编译时的错误情况。

8.5.2 #line 指示

`#line` 是用来改变程序行编号方式和文件名的指示。程序行一般是按 1, 2, 3, …… 的方式来编号的。



`#line` 指示有两种形式。先来看第一种形式：

```
#line 行号
```

其中，行号必须是介于 1~32 767 之间的整数。此指示导致程序后面的行被编号为 `n`、`n+1`、`n+2` 等。

`#line` 指示的第二种形式如下：

```
#line 行号 "文件名"
```

其中，行号和文件名说明来源的行号和文件名，指示后面的行会被认为是来自文件，行号由 `n` 开始。

`#line` 指示的一种作用是改变 `__LINE__` 宏或 `__FILE__` 宏的值。而且，大多数编译器使用来自 `#line` 指示的信息都会产生出错消息。实际上，程序员并不经常使用 `#line` 指示。它主要用于那些产生 C 代码作为输出的程序。比如前面介绍的 `-v` 选项就会使预处理器调用 `#line` 指示，从而按照源程序的行号和文件名处理。

8.5.3 `#pragma` 指示

`#pragma` 指示为编译器执行某些特殊操作提供了一种方式。对非常大的程序或需要使用特定编译器的特殊功能的程序非常有用。

`#pragma` 指示的一般形式如下：

```
#pragma 记号
```

其中，记号表示了一条编译器需要服从的命令。

一些编译器允许 `#pragma` 指示所包含的不仅是简单的命令。特别是有些编译器允许 `#pragma` 指示带参数，例如：

```
#pragma data(heap_size =>1024, stack_size =>2048)
```

`#pragma` 指示中出现的命令集在不同的编译器上不尽相同。必须查阅所使用的编译器的文档来了解哪些命令是可以使用的，以及这些命令的功能。而且，如果 `#pragma` 指示包含了无法识别的命令，则编译器必须忽略这些 `#pragma` 指示，不允许产生出错消息。

8.6 “#”和“##”运算符

本节来介绍 C 语言中两个与预处理有关的运算符，即“#”运算符和“##”运算符。与之前介绍的运算符相比，使用并不广泛，所以并不被许多程序员熟知。不过，正所谓“艺多不压身”，学习这两个运算符还是能帮助我们完成一些实际的工作的。

8.6.1 “#”运算符

井号“#”在 C 语言中也是一种运算符，编译器仅允许“#”运算符出现在带参数的宏的替换文本中，将跟在其后的参数转换成一个字符串常量。有时将这种用法的“#”运算符称为字符串化运算符。

前面介绍过用带参宏定义简化常用输出的程序，但是如果想在输出时显示变量名该怎么办？下面的程序清单使用宏定义完成这个功能：

```
#define PF_INT(i) printf("#i"="%d\n",i)

void main()
{
    int x=100;
    PF_INT(x)
}
```

预处理器处理后输出语句将变为:

```
printf("x"="%d\n",x);
```

因为 C 语言中两个相邻的字符串常量会合并,所以上面语句相当于:

```
printf("x=%d\n",x);
```

8.6.2 “##” 运算符

“##”也是一种运算符,是将两个运算对象连接到一起,也只能出现在带参宏定义的替换文本中。也可以认为是将两个运算对象“粘”在了一起。预处理器将出现在“##”两侧的运算对象合并成一个符号。当运算对象有一个是宏参数时,“粘合”会在宏替换时发生。来看下面的宏定义:

```
#define NUM(h,t,u) h##t##u
```

假设其中的参数 h、t、u 分别表示一个十进制数的百位、十位和个位上的数字,则下面的语句给变量 x 赋值为 123:

```
x=NUM(1,2,3);
```

因为“##”运算符很少有程序员会知道,绝大多数程序员也从来没用过它。所以读者只需知道有此运算符即可。

8.7 预处理实例

俗话说:“光说不练假把式”,本节就通过一个简单的计算器程序来对所介绍的预处理指示进行练习,主要练习 3 种预处理器的基本功能。

8.7.1 简单计算器程序

本小节实现一个简单计算器程序,此程序可以实现加、减、乘、除、求余、平方、立方、开平方运算。limt.h 文件清单如下所示:

```
#ifndef LIMT_H
#define LIMT_H

#define TO_LONG(x) (long)(x) //将参数转换为 long 型
#define IS_PLUS(x) (x>=0? 1:0) //判断参数是否为正数
#define CHECK_ZERO(divisor) (divisor) == 0? 1:0 //判断除数是否为 0 的宏

#endif
```

myio.h 文件清单如下所示:



```
#ifndef MYIO_H
#define MYIO_H

#ifdef __cplusplus
#include <iostream>
#else
#include <stdio.h>
#endif

#define CLUE_OPE    printf("请输入一个运算符, +表示加法, -表示减法, *表示乘法, \
                        /表示除法, \
%%表示求余, 2 表示乘方, 3 表示立方, s 表示开平方\n")    //输出提示信息的宏
#define CLUE_TWO    printf("请输入两个运算对象:\n")        //提示输入两个运算对象的宏
#define CLUE_ONE    printf("请输入一个运算对象:\n")        //提示输入一个运算对象的宏
#define CLUE_ZERO    printf("你输入的被除数为 0, 请重新输入:\n")    //提示被除数不为 0 的宏
#define CLUE_PLUS    printf("开平方的数必须是正数, 请重新输入:\n")    //提示被开方数必须为正
#define IN_OP(x)    x=getchar()    //输入运算符的宏
#define IN_O(x)    scanf("%lf",&x)    //输入一个数据的宏
#define IN_T(x,y)    scanf("%lf%lf",&x,&y)    //输入两个数据的宏
#define OUT_D(x)    printf("计算结果为:%f\n",x)    //输出 double 型数据的宏
#define OUT_L(x)    printf("计算结果为:%ld\n",x)    //输出 long 型数据的宏

#endif
```

count.h 文件清单如下所示:

```
#ifndef COUNT_H
#define COUNT_H

#include <math.h>

#define SUM(x,y) ((x)+(y))    //计算加法的宏
#define SUB(x,y) ((x)-(y))    //计算减法的宏
#define MUL(x,y) ((x)*(y))    //计算乘法的宏
#define DIV(x,y) ((x)/(y))    //计算除法的宏
#define MOD(x,y) ((x)%(y))    //计算余数的宏
#define SQUARE(x) ((x)*(x))    //计算乘方的宏
#define CUBE(x) ((x)*(x)*(x))    //计算立方的宏
#define Sqrt(x) (sqrt(x))    //计算开方的宏

void count(char);

#endif
```

demo.c 文件清单如下所示:

```
#include "limt.h"
#include "myio.h"
#include "count.h"

int main(void)
```



```
{
    char c;

    CLUE_OPE;
    count(IN_OP(c));

    return 0;
}

void count(char c)
{
    double x,y;
    switch(c)
    {
        case '+':          //进行加法运算
            CLUE_TWO;
            IN_T(x,y);
            OUT_D(SUM(x,y));
            break;
        case '-':          //进行减法运算
            CLUE_TWO;
            IN_T(x,y);
            OUT_D(SUB(x,y));
            break;
        case '*':          //进行乘法运算
            CLUE_TWO;
            IN_T(x,y);
            OUT_D(MUL(x,y));
            break;
        case '/':          //进行除法运算
            CLUE_TWO;
            IN_T(x,y);
            while(CHECK_ZERO(y))
            {
                CLUE_ZERO;
                IN_T(x,y);
            }
            OUT_D(DIV(x,y));
            break;
        case '%':          //进行求余运算
            CLUE_TWO;
            IN_T(x,y);
            while(CHECK_ZERO(y))
            {
                CLUE_ZERO;
                IN_T(x,y);
            }
            OUT_L(MOD(TO_LONG(x),TO_LONG(y)));
            break;
        case '2':          //进行平方运算
```



```
        CLUE_ONE;
        IN_O(x);
        OUT_D(SQUARE(x));
        break;
    case '3':          //进行立方运算
        CLUE_ONE;
        IN_O(x);
        OUT_D(CUBE(x));
        break;
    case 's':          //进行开平方运算
        CLUE_ONE;
        IN_O(x);
        while(!IS_PLUS(x))
        {
            CLUE_PLUS;
            IN_O(x);
        }
        OUT_D(SQRT(x));
        break;
    }
}
```

8.7.2 程序分析

1. 程序整体结构

可以看出, 此程序包括一个源文件 `demo.c` 和 3 个头文件: `limt.h`、`count.h` 和 `myio.h`。程序整体的结构如图 8.8 所示。

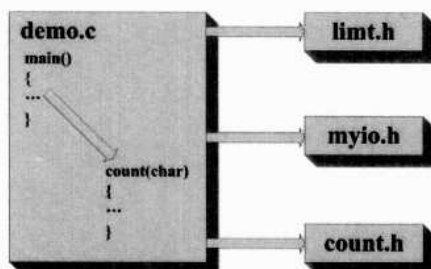


图8.8 程序整体结构图

2. 各个文件的作用

- ☑ `limt.h`: 此头文件的作用是提供一些对于运算对象的限制和转换, 包括对被除数为 0 的限制、开平方数是正数的限制, 以及将求余运算的运算对象转换为 `long` 型。
- ☑ `count.h`: 此头文件的作用是提供各种形式运算的宏定义以及运算函数原型。
- ☑ `myio.h`: 此头文件的作用是提供各种形式的输入与输出操作, 可称为自定义输入/输出库。
- ☑ `demo.c`: 此文件的作用是演示各种运算。它是简单计算器程序的具体实现, 使用 3 个头文件中的定义的宏完成简单计算器的功能。

3. 优点

采用此种文件结构有以下 3 个优点。

☑ 提高程序扩展性

将对运算对象的限制、输入/输出、运算方法放在不同的头文件中，而将具体实现放在一个源文件中的这种方式可以提高程序的扩展性。假设此程序现在要运行在某个特殊的机器上，而需要对参与运算的对象进行大小的限制，那么只需在 `limt.h` 中增加一些限制即可。如果需要此计算器执行诸如 `sin()`、`cos()` 之类的三角函数运算，则只需在 `count.h` 中增加一些运算的宏即可。如果需要增加 I/O 操作时也只需在 `myio.h` 中增加代码即可。当然，要使用到这些扩展的内容时还要更新 `demo.c`，或者根据不同的应用重新写一个 `demo.c`。

☑ 提高程序可维护性

与提高扩展性的原理一样，如果程序的某个地方需要修改，可根据修改的内容对运算对象的限制、输入/输出、运算方法 3 个方面选择修改其相应的头文件。这样就大大地提高了程序的可维护性，使修改程序变得简单、清晰。

☑ 提高程序的封装性

所谓封装性是指将程序的某些代码包装在一起，从而达到一种保护功能。例如将一些语句封装在函数里，使用时只调用函数，而不具体使用其中的某个语句。

假设现在 3 个头文件是由不同的程序员开发的，为了保护自己的程序不被他人修改，采用目前这种将特定功能放在特定头文件中只供引用是一种很好的方式。而对于使用者来说，只需要用 `#include` 指示来包含这些文件，完全的“拿来主义”，而不用去管具体的实现过程。这样就能使大型的程序模块化，每个开发者只需做好自己的模块即可。

当然，与其他面向对象的语言（如 C++、Java）相比，C 语言的封装性并不出众，所以就要求在开发程序时更好地设计程序的框架，从而尽可能地提高其封装性。

8.7.3 程序中的预处理

程序中多次包含对宏定义、条件编译和文件包含这 3 种基本预处理指示的使用，下面对这 3 种指示的应用给予简单分析。

1. 对宏定义的使用

3 个头文件根据各自实现的功能使用了大量的宏定义。从而简化了很多操作。例如，源程序中只使用了一个简单的宏就实现了用 `printf` 语句书写可能需要两行语句的输出，这不仅大大提高了开发程序的效率。



注意：此程序旨在演示宏等预处理指示的使用方法，对于各种预处理指示的使用，特别是 `myio.h` 中对于宏的使用可以说是达到了“泛滥”的程度，这其实并不是一种好的编程方法。运行程序后可以发现，程序比不使用宏可能会慢了很多。所以，请读者在编写程序时，要合理地使用宏和其他预处理指示。

2. 对条件编译的使用

3 个头文件都使用了条件编译，旨在防止重复包含。在此，只对 `myio.h` 中下述的条件编译给予解释：




```
#ifndef __cplusplus
#include <iostream>
#else
#include <stdio.h>
#endif
```

其意义：如果包括 C++ 的标准库，就包含 C++ 的标准库头文件 “iostream”，否则，就包含 C 的标准库头文件 “stdio.h”。

可见，使用条件编译不仅可以使程序选择性地运行在不同的操作系统或不同的编译器上，甚至可以运行在不同的语言环境中。

3. 对文件包含的使用

对于文件包含的使用主要包括对于标准库的使用和对于自定义头文件的使用。遵循前面介绍的规则，对于 “stdio.h” 和 “math.h” 这类标准库文件采用尖括号的方式来包含，而对于自己定义的头文件 “count.h”、“limt.h”、“myio.h” 采用双引号的方式来包含。

 **提示：**请读者自己实现不使用预处理指示（不包括对于标准库头文件的包含）的简单计算器的版本，来比较两个版本程序的结构、开发效率，以及运行效率上有什么差别。





第 9 章

结构体与共用体



C 语言的数据类型丰富，除了提供基本数据类型（如整形、实型、字符型等数据类型）外，还提供了一些构造类型。构造类型包括前面介绍的数组类型，以及本章将要介绍的结构体类型和共用体类型。另外，有一种基本类型——枚举类型，因使用形式类似于构造类型，也将在本章中予以介绍。

虽然 C 语言的数据类型已经相当丰富，但有时并不能满足编程的需要，所以 C 语言还提供了用户自定义类型，从而使数据类型更加丰富多彩。本章也将对用户自定义类型进行介绍。





9.1 结构体

数组只能存储一种数据类型，如果要处理的数据中包含多种数据类型，数组就“无能为力”了，对此引出了一种新的数据类型——结构体（struct）。

本节从结构体的概念入手，详细介绍结构体变量的定义、引用、初始化、结构体数组、结构体指针、结构体和函数以及位域。

9.1.1 什么是结构体

在现实生活中，一个事物往往具有多个属性。例如，一辆汽车的基本信息包括汽车品牌（字符型）、车型（字符型）、马力（整型）等。一个班级中学生基本信息包括学号（字符型）、姓名（字符型）、性别（字符型）、年龄（整型）、班级（字符型）、成绩（整型或者浮点型）等。显然不能用数组来表示这些数据，因为数组的元素类型必须是一致的。在此情况下，可以使用 C 语言提供的结构体数据类型，使管理数据更加方便，而且还便于数据的封装与隐藏。

结构体是由一系列具有相同类型或不同类型的数据构成的数据集合，简称结构。在 C 语言中，可以定义结构体类型，将多个相关的变量包装成一个整体使用。在结构体中的变量，可以是相同、部分相同或完全不同的数据类型。在 C 语言中，结构体中不能包含函数类型。

结构体定义的一般形式为：

```
struct 结构体类型名
{
    数据类型 成员 1;
    数据类型 成员 2;
    ...
    数据类型 成员 N;
};
```

其中，struct 为关键字，结构体类型名是用户定义的合法 C 语言标识符，“{ }”中是组成该结构体的成员。成员的数据类型可以是 C 语言所允许的任何数据类型（除函数类型）。

9.1.2 结构体实例——《水浒传》中的一百单八将

在现实生活中可以用结构体表示的事物比比皆是，例如，我国四大名著之一——《水浒传》，其中包括性格各异的 108 个人物，如果现在要使用一种 C 语言的数据类型表示这些人物，结构体无疑是最佳选择。下面用一个名为 hero 的结构体来表示：

```
struct hero
{
    int number;           //排名
    char sex;             //性别
    char star_name[20];    //星名
    char name[20];         //名称
    char nickname[20];     //绰号
    char position[20];     //职业
    char weapon[20];       //使用武器
};
```

这个结构体如图 9.1 所示（括号中表示此结构体类型变量可能的取值）。

9.1.3 结构体类型与结构体变量

知道了结构体类型后，下面介绍一下结构体变量。在介绍结构体变量之前，先来看下面的语句：

```
int i;
```

此语句定义了一个变量 `i`，其类型为 `int`。同样，`struct` 作为 C 语言的一种数据类型，当然可以与其他类型一样，定义一个 `struct` 类型的变量，例如：

```
struct hero songjiang;
```

此语句定义了一个名为 `hero` 的结构体类型变量 `songjiang`，`hero` 是这个结构体类型名。即结构体变量 `songjiang` 是结构体 `hero` 的一个实例，`hero` 是名为 `songjiang` 的数据的类型。同样，也可以定义 `hero` 类型的变量 `luda`、`likui`、`linchong`：

```
struct hero luda;  
struct hero likui;  
struct hero linchong;
```

这 4 个结构体变量如图 9.2 所示（假设各个结构体变量已被初始化）。

编译器并不为结构体类型名分配存储空间，但是一旦定义了结构体变量，就要为其分配存储空间。即上面的类型名 `hero` 并没有占用计算机内存中的任何空间，但是编译器会给变量 `songjiang`、`luda`、`likui`、`linchong` 分配一定的内存空间。

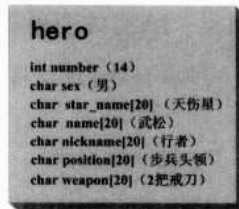


图9.1 结构体hero

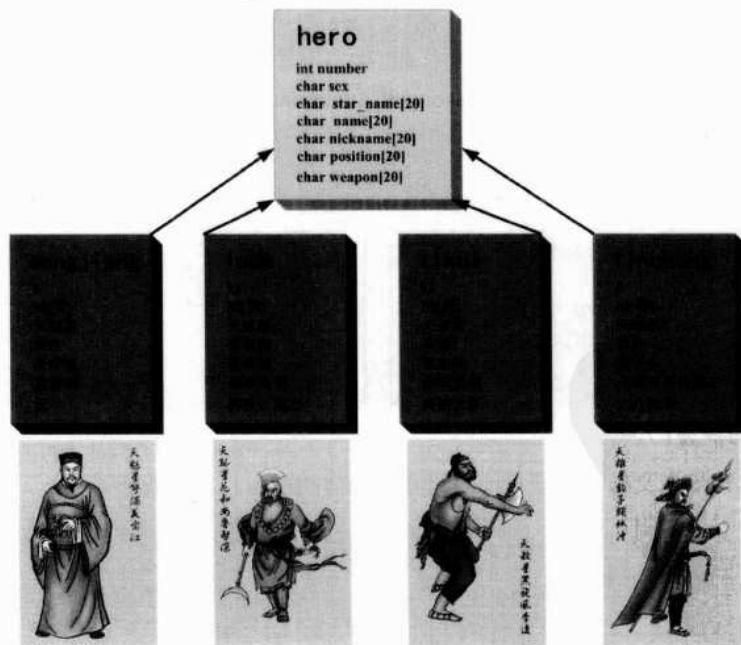


图9.2 结构体类型与结构体变量



9.1.4 结构体变量的定义

和基本类型的变量一样，结构体变量也要先定义，后使用。在 C 语言中，结构体变量的定义有 3 种形式。

1. 第一种形式

这种形式先声明结构体类型，再定义结构体变量。其一般形式为：

```
struct 结构体类型名
{
    数据类型名 1  成员名 1;
    ...
    数据类型名 n  成员名 n;
};
struct 结构体类型名 变量名列表;
```

前面在定义结构体变量 songjiang、luda、likui、linchong 时都是采用这种形式，下面再使用此形式定义两个 hero 类型的变量 wusong 和 sunerniang：

```
struct hero wusong, sunerniang; //定义 wusong 和 sunerniang 两个结构体变量
```

2. 第二种形式

这种形式在声明类型的同时定义变量。其一般形式为：

```
struct 结构体类型名
{
    数据类型名 1  成员名 1;
    ...
    数据类型名 n  成员名 n;
} 变量名列表;
```

同样是定义两个 hero 类型变量 wusong 和 sunerniang，用此种形式定义如下：

```
struct hero
{
    int number;           //排名
    char sex;             //性别
    char star_name[20];    //星名
    char name[20];         //名称
    char nickname[20];     //绰号
    char position[20];     //职业
    char weapon[20];       //使用武器
} wusong, sunerniang;     //定义 wusong 和 sunerniang 两个结构体变量
```

3. 第三种形式

这种形式直接定义结构体变量，而没有结构体类型名。其一般形式为：

```
struct
{
    数据类型名 1  成员名 1;
    ...
    数据类型名 n  成员名 n;
} 变量名列表;
```

同样是定义两个结构体类型变量 `wusong` 和 `sunerniang`，用此种形式定义如下：

```
struct
{
    int number;           //排名
    char sex;             //性别
    char star_name[20];   //星名
    char name[20];        //名称
    char nickname[20];    //绰号
    char position[20];    //职业
    char weapon[20];      //使用武器
} wusong, sunerniang;    //定义 wusong 和 sunerniang 两个结构体变量
```

9.1.5 定义结构体变量注意事项

在定义结构体变量时应注意以下几点：

- ☐ 如果使用第三种形式定义结构体变量，因为此形式没有给出结构体类型名，所以只能在定义结构体的同时定义结构体变量，比如上面使用第三种形式定义的变量 `wusong` 和 `sunerniang` 就遵循了此规则。
- ☐ 结构体的成员也可以是一个结构体变量。

例如，下面给结构体 `hero` 增加一个成员 `death_day`：

```
struct date
{
    int month;
    int day;
    int year;
};

struct hero
{
    ...
    struct date death_day; //死亡时间
    ...
} wusong, sunerniang;    //定义 wusong 和 sunerniang 两个结构体变量
```

上面的程序先定义了一个 `struct date` 类型，包括 3 个成员：`month`（年）、`day`（月）、`year`（日）。然后定义了 `struct hero` 类型，其中的成员 `death_day` 既是结构体 `hero` 的成员，又是 `struct date` 类型的变量。这两个结构体的关系如图 9.3 所示。

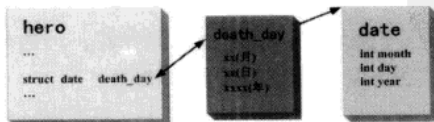


图9.3 结构体类型的结构体成员

9.1.6 结构体变量的初始化

和 C 语言中其他变量一样，在定义结构体变量时可以进行初始化操作，其一般形式为（“[]”中的内容可以省略）：



```

struct [结构体类型名]
{
    数据类型名 1    成员名 1;
    ...
    数据类型名 n    成员名 n;
} 结构体变量 = {初始数据};

```

例如，可以在定义 hero 类型变量 wusong 时对其进行初始化：

```

struct hero
{
    int number;           //排名
    char sex;             //性别
    char star_name[20];    //星名
    char name[20];        //名称
    char nickname[20];    //绰号
    char position[20];    //职业
    char weapon[20];      //使用武器
} wusong={14, 'm', "天伤星", "武松", "行者", "步兵头领", "2 把戒刀"};
//定义了结构体变量 wusong 并初始化

```

定义结构体变量 wusong 但没有初始化时，编译器会给每个成员一个默认值，初始化后就赋予初始化时的赋值，如图 9.4 所示。



图9.4 结构体变量及其初始化

注意：对结构体变量进行初始化时，必须按照每个成员的顺序和类型一一对应地赋值，少赋值、多赋值以及类型不符都可能引起编程错误。

9.1.7 结构体变量的引用

在定义了结构体变量后，就可以引用这个变量。所谓引用结构体变量就是使用结构体变量或结构体变量的成员进行运算或者其他操作。

1. “.” 运算符

在 C 语言中，“.” 也是一个运算符，叫做成员运算符。一般和结构体或共用体变量名称一起使用，用来指定结构体或共用体变量的成员。例如：

```
linchong.nickname;
```

用来指定结构体变量 linchong 的成员 nickname。

2. 结构体变量成员的引用

C 语言允许引用结构体变量的成员完成某种操作。其一般形式为：

```
结构体变量名.成员名
```

例如，对于上面定义的结构体变量 `wusong`，可以这样引用其成员：

```
wusong.number, wusong.sex;    //应用结构体变量
wusong.number+1;              //对结构体进行运算
printf("%d\n", wusong.number); //输出结构体成员的值
```

也可对定义了结构体变量的成员赋值：

```
sunerniang.number=103;
sunerniang.nickname="母夜叉";
```

3. 整个结构体变量的引用

C 语言允许对两个相同类型的结构体变量之间进行整体赋值。例如，下面将 `wusong` 的值赋给与其类型相同的变量 `zhangsan`：

```
zhangsan=wusong;
```

9.1.8 引用结构体变量注意事项

在引用结构体变量时，需注意以下几点：

☑ 必须先定义结构体变量，才能对其进行引用。

当开发大型程序时，引用未定义的结构体变量的错误可能会时常出现，不过，幸运的是，编译器会发现此错误，程序将无法编译通过。但是，养成良好的编程习惯还是必须的，毕竟调试错误也是一件很麻烦的事情。

☑ 如果一个结构体变量的成员又是一个结构类型，引用时要用成员运算符逐级遍历到最底层的成员。

例如，下面对前面定义的 `hero` 类型变量 `wusong` 的成员 `death_day` 的结构体成员进行引用：

```
wusong.death_day.month
wusong.death_day.day
wusong.death_day.year
```

其中，`wusong` 为第一级的结构体变量名，`death_day` 既为 `wusong` 的成员，又为第二级结构体变量名，`month`、`day`、`year` 为 `death_day` 的成员。

☑ 结构体变量成员可以像普通变量一样参与各种运算或其他操作。例如：

```
songerniang.sex='w';          //对结构体变量成员进行赋值操作
songerniang.number++;          //对结构体变量成员进行自增运算
sunerniang.number>wusong.number; //比较两个结构体变量成员
```

☑ 可以引用结构体变量地址，也可以引用结构体变量成员的地址。

例如，下面都是合法的语句：

```
//以十六进制数的形式输出结构体变量 wusong 的地址
printf("%x",&wusong);
//以十六进制数的形式输出结构体变量 wusong 成员 number 的地址
printf("%x",&wusong.number);
//从键盘上输入字符串给 sunerniang 变量的成员 sex 赋值
scanf("%c",&sunerniang.sex);
```

☑ 不能对结构体变量整体进行诸如输入/输出的操作。

例如，下面的语句是非法的：



```
scanf("%d%c%s%s%s%s", &sunerniang);
```

☐ 结构体成员名可以与程序中的变量名相同，但两者是完全不同的两个变量。

例如，下面的程序清单说明了局部变量和结构体中的成员变量可以同名，但不代表同一个变量：

```
//声明结构体类型
struct hero
{
    int number;           //排名
    char sex;             //性别
    char star_name[20];   //星名
    char name[20];        //名称
    char nickname[20];    //绰号
    char position[20];    //职业
    char weapon[20];      //使用武器
};

void main()
{
    int number=100;       //局部变量
    struct hero sunerniang;
    sunerniang.number=103; //结构体成员
}
```

此程序中两个同名的变量如图 9.5 所示。

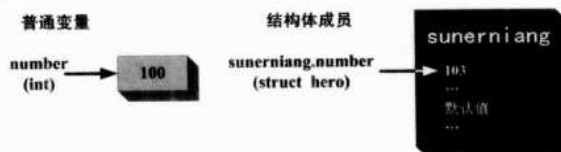


图9.5 同名的不同变量

9.1.9 结构体数组

1. 什么是结构体数组

一个结构体变量中可以存放若干个相关的数据。如果有一组或者一批类型相同的结构体数据要处理，显然应该使用结构体作为数组的成员，这就是结构体数组。与一般数组不同的是，结构体数组的每个元素都是结构体类型数据，这些元素又分别包括结构体的各个成员。结构体数组及元素，如图 9.6 所示。

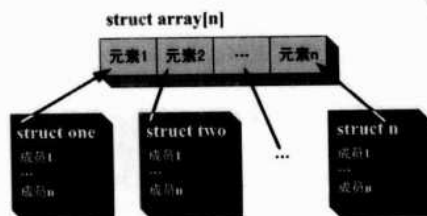


图9.6 结构体数组及元素

2. 定义结构体数组

和普通数组相同，结构体数组也必须先定义，后使用。定义结构体数组的方法与定义结构体变量的方法相同，区别在于需要说明变量为数组类型即可。其一般形式为：

```
struct 结构体类型名
{
    数据类型名1 成员名1;
    ...
    数据类型名n 成员名n;
};
struct 结构体类型名 数组变量名列表;
```

也可以使用类似于其他两种定义结构体变量的方式来定义结构体数组，此处就不再叙述，留给读者去思考。

下面使用第二种方式来定义一个表示一百单八将中的3位女性英雄的数组 `women`：

```
struct hero
{
    int number;           //排名
    char star_name[20];    //星名
    char name[20];         //名称
    char nickname[20];     //绰号
    char weapon[20];       //使用武器
} women[3];
```

数组 `women` 共有3个元素，其元素类型都为 `struct hero`。而每个数组元素又包含5个成员，如图9.7所示。

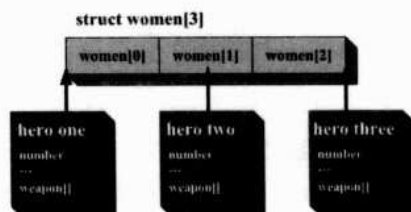


图9.7 结构数组 `women` 及元素

3. 初始化结构体数组

和C语言中的其他类型数组一样，定义结构体数组时也可以初始化。其一般形式为：

```
struct 结构体类型名
{
    数据类型名1 成员名1;
    ...
    数据类型名n 成员名n;
};
struct 结构体类型名 数组变量名列表 = {初值列表};
```

或者：

```
struct [结构体类型名]
{
```




```
数据类型名 1 成员名 1;
...
数据类型名 n 成员名 n;
} 数组变量名列表 = {初值列表};
```

其中，“[]”中的内容可以省略。

下面就来初始化前面定义的结构体数组 `women[3]`，数组 `women` 及其元素如图 9.8 所示。

```
women[3]={{59, 'W', "地慧星", "扈三娘", "一丈青", "日月双刀"},
           {101, 'W', "地阴星", "顾大嫂", "母大虫", "刀"},
           {103, 'W', "地壮星", "孙二娘", "母夜叉", "刀"}};
```

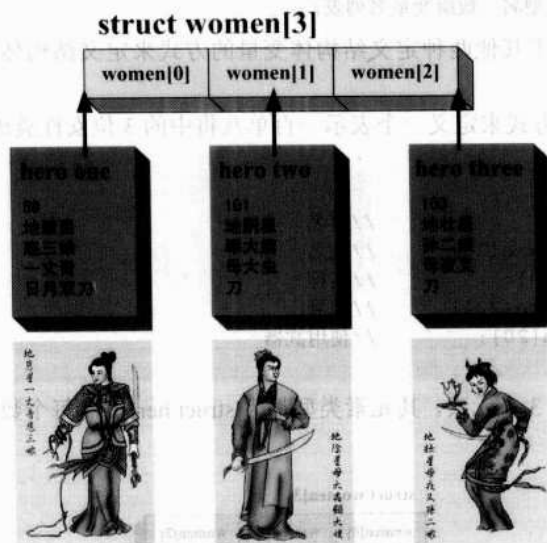


图9.8 结构体数组women及元素

4. 不指定元素个数的结构体数组定义

定义结构体数组时，元素个数也可以不指定，但必须在定义时进行初始化。其一般形式为（“[]”中的内容可以省略）：

```
struct 结构体类型名
{
    数据类型名 1 成员名 1;
    ...
    数据类型名 n 成员名 n;
} 数组变量名列表 = {{...}, {...}, {...}, {...}};
```

或：

```
struct 结构体类型名
{
    数据类型名 1 成员名 1;
    ...
    数据类型名 n 成员名 n;
};
struct 结构体类型名 数组变量名列表 = {{...}, {...}, {...}, {...}};
```


在编译时,系统会根据初始化值的个数自动确定数组元素的个数。可以简单地认为,元素个数为花括号数量(按对计算)减1。

5. 引用结构体数组变量

结构体数组变量的引用一般是指对数组元素成员的引用,和普通变量类似,只不过引用的对象是数组的元素。例如,下面是对 `women` 数组元素的正确引用:

```
women[0].number+1;  
printf("%s",women[2].nickname);
```

第一条语句引用结构体数组 `women` 的第一个元素的成员 `number` (类型为 `int`) 并进行加1运算,如果按照本节中对其初始化时的赋值,此表达的值为60。

第二条语句输出了 `women` 的第3个元素的成员 `nickname`,如果按照本节中对其初始化时的赋值,将输出《水浒传》中一个大名鼎鼎的绰号“母夜叉”。

需要注意的是既然引用的对象是数组元素,那么数组下标不能等于或超过数组元素的个数。例如下面的引用是错误的:

```
women[3].number> 100//数组下标错误
```

9.1.10 指向结构体的指针

1. 结构体指针变量声明的一般形式

与一般变量一样,可以使一个指针变量指向结构体,从而形成结构体指针变量。其值是指向的结构体变量的首地址。通过结构体指针即可访问该结构体变量,这与数组指针和函数指针的情况相同。结构体指针变量声明的一般形式为:

```
struct 结构名 *结构指针变量名;
```

对前面定义的结构体 `hero`,可以使一个指针 `phero` 指向 `hero` 类型的某个变量:

```
struct hero *phero;
```

既然结构体指针变量也是一种结构体变量,那么也可以使用其他两种定义结构体变量的方式定义结构体指针变量。具体定义方式此处不再叙述。

2. 为何需要结构体指针变量

之所以引入结构体指针变量,出于以下几个原因:

☑ 更易于操作:

类似于数组指针比数组更易于操作一样(如排序问题),结构体指针比结构体本身更加的易于操作。

☑ 更强的通用性:

一些早期的C语言实现不支持将结构体变量作为参数传递给函数,但是结构体指针变量却可以。

☑ 丰富的数据表示:

C语言中的许多数据表示(如文件指针)都使用了指向结构体的指针。

3. 指针变量到底指向什么

与前面讨论的各类指针变量相同,结构指针变量也必须要先赋值后使用。赋值是把结构



变量的首地址赋予该指针变量，不能把结构名赋予该指针变量。

例如，下面对结构体指针变量 phero1 的赋值是正确的：

```
*phero1=&wusong; //将变量 wusong 的首地址赋值给指针变量 phero
```

但是下面的赋值是错误的：

```
*phero2=&hero; //错误
```

前面笔者已经强调过，结构体名和结构体变量是两个不同的概念，不能混淆。结构体名只能表示一个结构形式，编译系统并不分配内存空间。只有当某变量被定义为这种类型的结构体时，才为该变量分配存储空间。所以 &wusong 的形式是正确的，表示结构体变量 wusong 的首地址，而 &hero 的形式是错误的，因为系统没有为 hero 分配内存，也不存在 hero 的首地址，如图 9.9 所示。

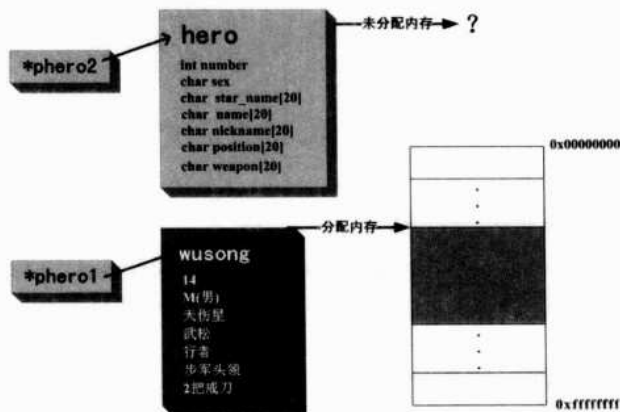


图9.9 结构体指针变量的内存分配

4. 访问成员

与其他结构体变量一样，可以使用“.”运算符访问结构体指针变量的成员，其一般形式为：

```
(*结构指针变量).成员名
```

例如，对前面定义的变量 phero 的成员 number 进行访问：

```
(*phero).number;
```

因为成员运算符“.”和指针运算符“*”是同一优先级的运算符，但其结合顺序是从右到左的，所以括号运算符“()”必不可少，即下面的形式是错误的：

```
*phero.number;
```

为了防止此类错误，C 语言还提供了一种访问结构体成员的方法，就是使用成员指针运算符“->”，使用“->”运算符访问结构体成员的一般形式如下：

```
结构指针变量->成员名
```

例如，下面用这种方式访问结构体指针变量 phero 的成员 name：

```
*phero=&linchong;  
Phero->name;
```

等价于:

```
(*phero).name;  
linchong.name;
```

9.1.11 结构体与函数

在 ANSI C 中, 结构体指针变量可以作为参数传递给函数, 同样, 函数也可以返回一个结构体指针变量。而且, 较新的 ANSI C 允许将结构体变量作为函数的参数和返回值。将结构体变量作为函数的参数和返回值与其他变量的规则相似, 本小节不再叙述。需要特别注意的是作为参数或返回值的是结构体变量还是结构体指针变量。

1. 结构体变量和结构体指针变量作为函数返回值

各种数据类型都可以作为函数返回值的类型。例如, 基本数据类型 `int`、`float` 和数组类型, 当然也可以是结构体类型。可作为函数返回值的结构体类型可以是一个结构体变量或结构体指针变量。下面给出这两种方式的函数原型:

```
struct struct_fuction(void);  
struct * struct_fuction(void);
```

貌似这两个函数原型的区别就是一个多一个指针运算符“*”, 另外一个则没有。从形式上看的确如此, 而其实质上的区别就在于一个返回的是指针对象, 一个返回的是结构体对象。

例如, 有两个变量, 一个为结构体变量, 一个为结构体指针变量, 二者都需要初始化一些必要的信息以便使用, 所以此处提供了两个初始化函数, 一个初始化结构体变量并返回结构体变量, 一个初始化结构体指针变量并返回结构体指针变量。

假如两个变量可以说话, 结构体变量会对初始化函数说: “兄弟, 帮我把衣服印上一点信息吧。” 初始化函数会说好的, 于是初始化函数会拿上喷枪将信息喷上结构体变量。然后结构体指针变量会对另外一个初始化函数说: “也给我的衣服印上点东西吧。” 初始化函数会说好的, 然后在别的衣服上印上信息后, 告诉结构体指针变量: “嗨, 兄弟, 你要的东西已经弄好了, 你可以自己去某某地方穿上。”

这样, 两个变量的实际对象——衣服上都有了一些信息, 只是印的方式不同而已, 一个是直接印上去, 一个则是套上一层印有信息的外壳, 如图 9.10 所示。

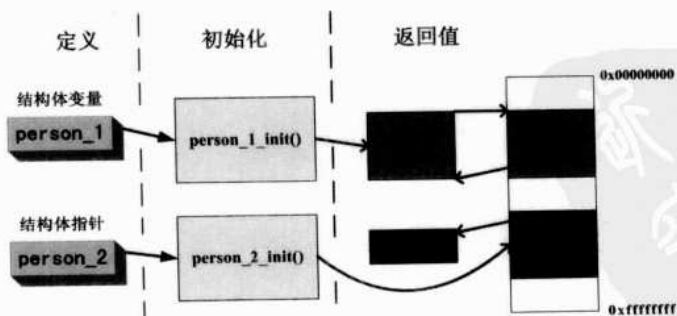


图9.10 结构体指针和结构体变量作为返回值

下面的结构体初始化程序清单说明了函数返回结构体和返回结构体指针的具体区别:



```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

//声明结构体类型
struct comm
{
    unsigned char name[10];
    unsigned char *address;
};
//定义结构体变量
struct comm person_1;
struct comm *person_2;

//函数声明
static struct comm person_1_init(const void *name, const void *address);
static struct comm * person_2_init(const void *name, const void *address);
static void struct_str_print(const unsigned char *struct_str, size_t length);
static size_t str_len(const void *str);
static void str_cpy(void *dest, const void *src, size_t len);

int main(void)
{
    //初始化结构体
    person_1 = person_1_init("Bright", "NewYeak");
    person_2 = person_2_init("Minos", "Hongkong");

    //打印初始化内容
    printf("person_1.name : %s\tperson_1.address : %s\n",
        person_1.name, person_1.address);
    printf("person_2->name : %s\tperson_2->address : %s\n",
        person_2->name, person_2->address);
    //打印函数返回值的字符串形式
    printf("person_1 : <");
    struct_str_print((unsigned char *)&person_1, sizeof(struct comm));
    printf(">\n");

    printf("person_2 : <");
    struct_str_print((unsigned char *)&person_2, sizeof(struct comm));
    printf(">\n");

    //回收堆空间
    free(person_1.address);
    free(person_2->address);
    free(person_2);

    //避免野指针
    person_1.address = NULL;
    person_2->address = NULL;
    person_2 = NULL;
```

```

    getchar();
    return 0;
}

/*
    结构体变量初始化函数，
    用形参来传递初始化数据，
    返回值为结构体或者结构体指针
*/
static struct comm person_1_init(const void *name, const void *address)
{
    assert((name != NULL) && (address != NULL));
    assert((str_len(name) + 1 <= 10) && (str_len(address) + 1 <= 100));

    person_1.address = (unsigned char *)malloc(sizeof(unsigned char) * 100);

    str_cpy(person_1.name, name, str_len(name));
    str_cpy(person_1.address, address, str_len(address));

    return person_1;
}

static struct comm * person_2_init(const void *name, const void *address)
{
    assert((name != NULL) && (address != NULL));
    assert((str_len(name) + 1 <= 10) && (str_len(address) + 1 <= 100));

    person_2 = (struct comm *)malloc(sizeof(struct comm));
    person_2->address = (unsigned char *)malloc(sizeof(unsigned char) * 100);
    str_cpy(person_2->name, name, str_len(name));
    str_cpy(person_2->address, address, str_len(address));

    return person_2;
}

/*
    打印固定大小的字符串，十六进制；形参为字符串指针，字符串长度；无返回值
*/
static void struct_str_print(const unsigned char *struct_str, size_t length)
{
    assert(struct_str != NULL);

    size_t str_idx;
    for (str_idx = 0; str_idx < length; ++str_idx)
    {
        printf("%x", *struct_str++);
    }
}

```




```
//计算字符串长度
static size_t str_len(const void *string)
{
    assert(string != NULL);
    size_t length = 0;

    const unsigned char *str = (const unsigned char *)string;
    while(*str++ != '\0')
        length++;


    return length;
}

/*
字符串复制
非完全版本，需注意数据溢出修改问题
*/
static void str_cpy(void *dest, const void *src, size_t len)
{
    assert((src != NULL) && (dest != NULL));

    size_t length = len + 1;
    unsigned char *dest_bp = (unsigned char *)dest;
    const unsigned char *src_bp = (const unsigned char *)src;

    while(length-- > 0)
        *dest_bp++ = *src_bp++;
}
```

从上面的程序可以看出，当返回值为结构体变量时，初始化函数返回该变量时是直接将该返回值初始化后对其赋值；而当返回值为结构体指针变量时，初始化函数返回该指针变量时是间接修改内存中的一个结构变量的值，并将该结构变量的初始地址返回。

 **说明：**选择结构体变量还是结构体指针变量作为函数返回值时，应该遵循具体应用来考虑使用间接写还是直接写，但是考虑到速度上的优势，大多数情况下使用间接写，即结构体指针变量作为返回值。

2. 结构体变量和结构体指针变量作为函数形式参数

结构体变量和结构体指针变量都可以作为函数的形式参数，先来看两者的函数原型：

```
struct struct_fuction(struct struct_pm,...);
struct * struct_fuction(struct *pstruct_pm,...);
```

上面两个函数原型中形式参数分别使用了结构体变量和结构体指针变量，其区别和返回值类似，一个传递的是普通变量，一个传递的是地址。

用两个变量的对话例子说明两者的区别：

假设有一个带有结构体变量形式参数的函数 A_INIT 和一个带有结构体指针变量形式参数的函数 B_INIT，并且还有结构体变量 A 和结构体指针变量 B。假设有一天两个变量 A 和 B 想去做一个发型，变量 A 去了理发店，而变量 B 则给理发店下了一个订单并且留下了地

址。变量 A 到达了理发店的时候得到了函数发型师 A_INIT 的接待, 变量 B 则在家里等待函数发型师 B_INIT 的上门服务。

函数发型师 A_INIT 首先用模型头复制了 A 原来的发型, 然后根据 A 需要的发型参数来对 A 的复制模型头进行发型制作, 做好了以后将 A 原来的发型用复制模型头的发型覆盖掉, 至于是怎么覆盖的, 留给读者一点遐想空间; 此时, 函数发型师 B_INIT 根据变量 B 所留的地址到达变量 B 的家里, 并马上根据变量 B 要求的发型参数直接在 B 的头上进行发型的制作, 然后变量 A 和变量 B 都有了新的发型。

上述例子比较形象地描述了函数对待结构体变量形参函数和结构体指针变量形参函数的区别, 其具体的区别如图 9.11 所示:

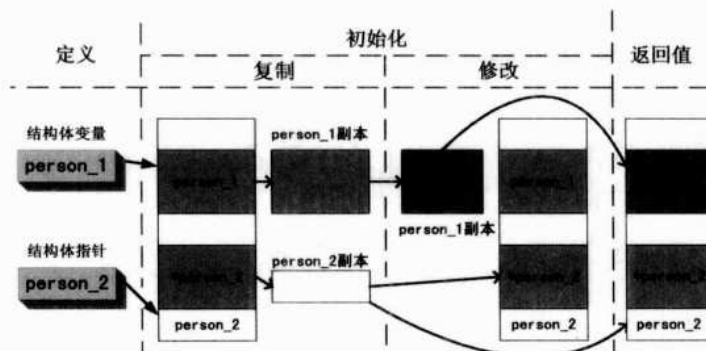


图9.11 结构体指针和结构体变量作为形参

下面的程序与前面的结构体初始化程序功能一致, 只不过使用了结构体变量和结构体指针作为函数形式参数, 下面给出程序清单:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

//声明结构体类型
struct comm
{
    unsigned char name[10];
    unsigned char *address;
};

//函数声明
static struct comm person_1_init(struct comm person,
    const void *name, const void *address);
static struct comm * person_2_init(struct comm *person,
    const void *name, const void *address );
static void struct_str_print(const unsigned char *struct_str, size_t length);
static size_t str_len(const void *str);
static void str_cpy(void *dest, const void *src, size_t len);

int main(void)
```




```
{
    //定义结构体变量
    struct comm person_1;
    struct comm *person_2;

    //初始化结构体
    person_1 = person_1_init(person_1, "Bright", "NewYeak");
    person_2 = person_2_init(person_2, "Minos", "Hongkong");

    //打印初始化内容
    printf("person_1.name : %s\tperson_1.address : %s\n",
           person_1.name, person_1.address);
    printf("person_2->name : %s\tperson_2->address : %s\n",
           person_2->name, person_2->address);
    //打印函数返回值的字符串形式
    printf("person_1 : <");
    struct_str_print((unsigned char *)&person_1, sizeof(struct comm));
    printf(">\n");

    printf("person_2 : <");
    struct_str_print((unsigned char *)person_2, sizeof(struct comm *));
    printf(">\n");

    //回收堆空间
    free(person_1.address);
    free(person_2->address);
    free(person_2);

    //避免野指针
    person_1.address = NULL;
    person_2->address = NULL;
    person_2 = NULL;

    getchar();
    return 0;
}

/*
    结构体变量初始化函数,
    用形参传递初始化数据,
    返回值为结构体或者结构体指针
*/
static struct comm person_1_init(struct comm person,
                                const void *name, const void *address)
{
    assert((name != NULL) && (address != NULL));
    assert((str_len(name) + 1 <= 10) && (str_len(address) + 1 <= 100));

    person.address = (unsigned char *)malloc(sizeof(unsigned char) * 100);
}
```

```

    str_cpy(person.name, name, str_len(name));
    str_cpy(person.address, address, str_len(address));

    return person;
}

static struct comm * person_2_init(struct comm *person,
    const void *name, const void *address)
{
    assert((name != NULL) && (address != NULL));
    assert((str_len(name) + 1 <= 10) && (str_len(address) + 1 <= 100));

    person = (struct comm *)malloc(sizeof(struct comm));
    person->address = (unsigned char *)malloc(sizeof(unsigned char) * 100);
    str_cpy(person->name, name, str_len(name));
    str_cpy(person->address, address, str_len(address));

    return person;
}

/* 打印固定大小的字符串，十六进制；形参为字符串指针，字符串长度；无返回值 */
static void struct_str_print(const unsigned char *struct_str, size_t length)
{
    assert(struct_str != NULL);

    size_t str_idx;
    for (str_idx = 0; str_idx < length; ++str_idx)
    {
        printf("%x", *struct_str++);
    }
}

// 计算字符串长度
static size_t str_len(const void *string)
{
    assert(string != NULL);
    size_t length = 0;

    const unsigned char *str = (const unsigned char *)string;
    while(*str++ != '\0')
        length++;

    return length;
}

```



```
/*
    字符串复制
    非完全版本，需注意数据溢出修改问题
*/
static void str_cpy(void *dest, const void *src, size_t len)
{
    assert((src != NULL) && (dest != NULL));

    size_t length = len + 1;
    unsigned char *dest_bp = (unsigned char *)dest;
    const unsigned char *src_bp = (const unsigned char *)src;

    while(length-- > 0)
        *dest_bp++ = *src_bp++;
}
```

从上面的程序可以看出，当结构体变量作为形参时，在调用该函数时首先复制整个结构体形成结构体副本，然后对副本进行修改，但并没有修改原来的结构体，再将副本作为返回值返回；当结构体指针作为形参时，在调用该函数时首先复制的是该结构体的地址，形成一个地址副本，然后使用这个地址副本对其指向的结构体变量进行修改，其实修改的还是原来地址所指向的结构体变量，最后用地址副本作为返回值。

9.1.12 位域

1. 什么叫位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占一个或几个二进制位。例如，在存放一个开关量时，只有 0 和 1 两种状态，用一个二进制位表示即可。而在使用 C 语言开发程序，特别是一些系统级或嵌入式程序时，内存往往是特别宝贵的。为了节省存储空间，并使处理简便，C 语言提供了一种数据结构，即“位域”或“位段”。

所谓“位域”是把一个字节中的二进制位划分为几个不同的区域，并说明每个区域的位数。每个区域有一个域名，允许在程序中按域名进行操作。这样一个字节的二进制位就可以表示几个不同的量。

这有点类似于时下流行的两截裤，如果把一整条裤子看作一个字节，两截裤就是把一条裤子分成两截，而位域则是把一个字节分成几截，每截独立地表示一个量，如图 9.12 所示。

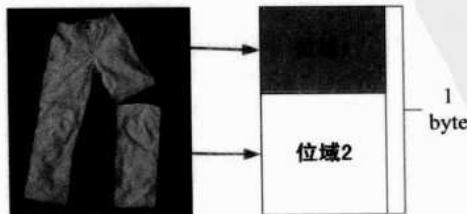


图9.12 两截裤与位域

2. 位域及位域变量的定义

位域的定义与结构定义相似，其形式为：

```
struct 位域结构名
{
    类型说明符 位域名 1:位域长度 1;
    ...
    类型说明符 位域名 n:位域长度 n;
};
```

同样，位域变量的定义与结构体变量定义的方式相同。可采用先定义位域后定义位域变量、同时定义或者直接定义位域变量 3 种方式。具体的形式此处就不再叙述。

例如，下面采用同时定义的方式定义一个位域变量 `field`：

```
struct byte_struct
{
    int x:8;
    int y:2;
    int z:6;
} field;
```

其表明 `field` 为 `byte_struct` 类型变量，共占 2 个字节。其中位域 `x` 占 8 位，位域 `y` 占 2 位，位域 `z` 占 6 位，如图 9.13 所示。

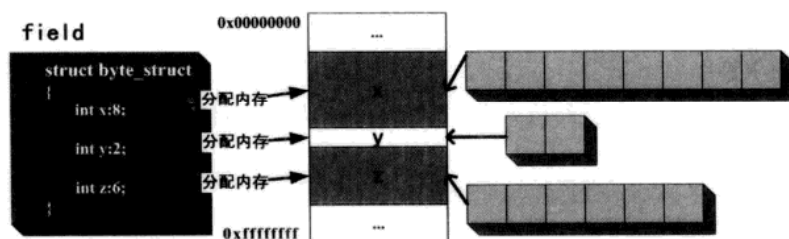


图9.13 位域变量的内存表示

3. 位域定义时注意事项

- 单个位域必须存储在同一个字节中，不能跨字节存储。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使用空域使某位域从下一单元开始。例如：

```
struct byte_struct
{
    unsigned a:6;
    unsigned :0; //空域
    unsigned b:4; //从下一单元开始存放
    unsigned c:4;
} two_byte;
```

其中，`a` 占第一字节的 6 位，后 2 位填 0 表示不使用，称为空域；`b` 从第二字节开始，占用 4 位，`c` 占用 4 位，如图 9.14 所示。

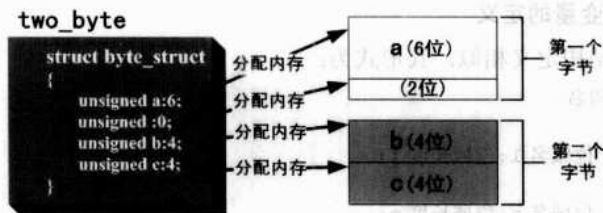


图9.14 位域变量的内存表示

位域的长度不能大于一个字节的长度，即不能超过 8 位。

例如，下面的定义是非法的：

```
struct bs
{
    unsigned a:2;
    unsigned b:10; //位域长度不能大于 8 位
};
```

位域可以无位域名，这时只用来做填充或调整位置，而不能使用。例如：

```
struct bs
{
    unsigned a:4;
    unsigned:4; //此 4 位不能使用
};
```

4. 位域的使用

位域从本质上说就是一种结构类型，不过其成员是按二进制位分配的。所以位域的使用和结构成员的使用相同。

下面的程序清单演示了普通位域变量和指针位域变量的使用：

```
#include <stdio.h>

//位域声明
struct byte_struct
{
    unsigned a:4;
    unsigned b:2;
    unsigned c:1;
};

void main()
{
    struct byte_struct bit; //定义普通位域变量
    //普通位域变量的使用
    bit.a=1;
    bit.b=7;
    bit.c=15;
    printf("%d,%d,%d\n",bit.a,bit.b,bit.c);
    struct byte_struct *pbit; //定义指针位域变量
    //指针位域变量的使用
```



```
pbit=&bit;
pbit->a=0;
pbit->b&=3;
pbit->c|=1;
printf("%d,%d,%d\n",pbit->a,pbit->b,pbit->c);
}
```

可以看出,位域变量的使用 and 结构体变量的使用从形式上看完全相同,只不过成员都是用二进制位来表示,所以会经常使用一些位操作。

9.2 共用体

C 语言除了提供结构体这种可包含多种类型数据的构造类型外,还提供了一种从形式上看和结构体堪称“孪生兄弟”的构造类型——共用体 (union)。

本节从共用体的概念、与结构体的异同、使用等方面进行详细介绍。

9.2.1 什么是共用体

在现实生活中,某些事物往往可以用多种方式来表述,各种方式的地位是平等的,都是从不同的侧面去反映这个事物。例如,古人用的字、名、号,都是对一个人的称谓,但是会根据不同的场合和情况使用其中的一种。又例如,用数字和用优、良、中、差都可以表示成绩,具体的使用可以根据成绩的种类不同选用其中的一种方式。

如果这些方式可以用同一种类型来表示,那么数组“勉强”能够用来对其进行存储,但浪费存储空间。而如果数据类型不同,数组就“无能为力”了。结构体似乎也能对其进行表述,但并不是最佳的方式。针对这种情况下,C 语言提供的共用体数据类型可以实现。

共用体,亦称联合,是有别于前述任何一种数据类型的特殊数据类型,它用来描述类型不相同的数据。与结构体不同的是:共用体对成员存储时采用覆盖技术,共享(部分)存储空间,成员被分配在同一段内存空间中。成员既可以具有相同的数据类型,也可以具有不同的数据类型。

共用体定义与结构体相似,其一般形式为:

```
union 共用体名
{
    数据类型 成员名 1;
    数据类型 成员名 2;
    ...
    数据类型 成员名 n;
};
```

例如,下面定义一个表示成绩的共用体:

```
union mark
{
    int score;        //表示分数
    char degree[4];   //表示等级
};
```

这个共用体如图 9.15 所示。

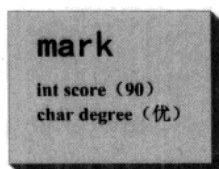


图9.15 共用体mark

9.2.2 共用体与结构体

共用体和结构体作为两种构造类型，从形式上看极其相似，但确实是两种不同的数据类型，可以说是“貌似神离”。下面从外部形式（外部）和内存使用（内部）两方面进行比较。

1. 外部的“情投意合”

共用体的定义、共用体变量的定义、共用体变量的引用等方面和结构体形式相同，可以简单地看做是用 union 关键字替代 struct 关键字。具体说明如下：

- ☑ 类似与结构体变量的3种定义方式，共用体变量也有3种形式的定义方式。即先定义共用体类型，再定义共用体变量；定义共用体类型的同时定义共用体变量；不含共用体类型名定义共用体变量。

例如，下面用第二种方式定义共用体变量 liming：

```
union mark
{
    int score;           //表示分数
    char degree[4];      //表示等级
} liming;
```

- ☑ 与结构体变量类似，可以使用“.”运算符引用其成员，但不能直接引用共用体变量。比如，下面对变量的引用是合法的：

```
liming.score=90;          //给共用体变量成员赋值
printf("%s\n",liming.degree); //输出共用体变量成员值
```

而下面的引用是非法的：

```
printf("%d,%s",liming);    //不能引用共用体变量
```

- ☑ 与结构体同样的是，编译器只为共用体变量分配内存空间，不为共用体名分配内存空间。

2. 内部的“各怀鬼胎”

共用体和结构体之所以是两种不同的类型，是因为在内存中的表示形式不同。可以简单地认为：共用体变量所占的内存长度等于其成员变量中所占存储空间最大的那个。而结构体变量其所占的内存长度等于其成员变量所占内存之和。

例如，下面定义一个结构体变量 sum：

```
struct
{
    int i;
    float f;
```



```
} sum;           //定义一个结构体变量
```

和一个共用体变量 max:

```
union
{
    int i;
    float f;;
} max;           //定义一个共用体变量
```

两个变量在内存中的表示如图 9.16 所示。

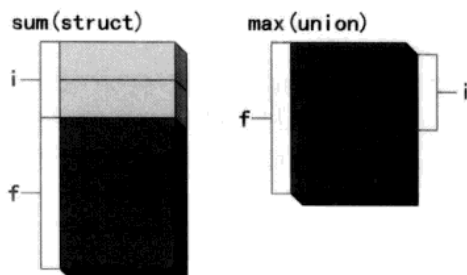



图9.16 结构体和共用体的内存表示

在图 9.16 中，每个框代表 1 个字节，假设 int 型数据占用 2 个字节的内存，float 型数据占用 4 个字节的内存。结构体变量 sum 所占的内存大小等于其成员 i 和 f 所占内存大小之和，即 6 个字节；而共用体变量 max 所占内存大小等于其成员中占内存最大的成员 f 所占内存大小，即 4 个字节。

 **注意：**在许多操作系统中，结构体的大小可能大于其内部成员大小之和，这是因为系统对数据存储进行了“内存对齐”，从而导致了存储上的间隙。所谓内存对齐是系统对数据类型的一种限制，要求某种类型对象的地址必须是某个值 n（通常是 2、4、8）的倍数，从而来简化处理器和存储器之间接口的硬件设计。

其中，Linux 的对齐策略是 2 字节数据类型，例如，short 的地址必须是 2 的倍数。而较大的数据类型（如 int、int*、float、double）则必须是 4 的倍数。而 Windows 的策略要求更为严格，要求任何 k 字节对象的地址必须是 k 的倍数。比如要求一个 double 类型对象的地址必须是 8 的倍数。所以开发时如果要用到结构体类型的大小，则需要查阅使用的操作系统、编译器的相关资料进行相应的处理。

9.2.3 共用体变量的初始化

共用体变量和结构体变量虽然很“貌似”，但是其初始化和结构体变量却截然不同。不能像结构体变量一样对共用体变量进行初始化。例如，下面想初始化共用体变量 liming，但这条语句是非法的：

```
union mark
{
    int score;           //表示分数
    char degree[4]; //表示等级
} liming={90, "优"}; //出错
```



要初始化共用体变量，只需初始化第一个成员，即赋予一个和第一个成员类型相同的常量。例如，下面是对共用体变量 `liming` 的正确初始化：

```
union mark
{
    int score;        //表示分数
    char degree[4];   //表示等级
} liming={90};
```

共用体变量不能作为赋值运算的左值和右值，除非是将一个共用体变量赋值给另一个和其类型相同的共用体变量。例如，下面的语句想通过赋值运算来给共用体变量赋值或将其赋值给另一个变量，不过这些都是非法的：

```
liming = 1;           // liming 为共用体变量，在赋值表达式中作为左值引用
x = liming;           // liming 为共用体变量，在赋值表达式中作为右值引用
```

不过，下面的语句将共用体变量 `liming` 赋值给和其类型相同的共用体变量 `zhangsan` 是可以编译通过的：

```
liming.score=100;
zhangsan=liming;      //将共用体变量 liming 赋值给共用体变量 zhangsan
```

9.2.4 使用共用体注意事项

由于共用体类型特殊的存储方式以及与结构体类型的“貌似神离”，因此在使用共用体变量时要特别小心，需注意以下几点：

☐ 共用体变量中的值是最后一次存放成员的值。

这是因为共用体的成员都存放在一块内存空间里，每次对成员值的修改，都相当于对这块内存空间的修改，就如同对变量多次赋值后变量的值为最后一次赋值一样。例如，下面对前面定义的共用体变量 `max` 的成员进行赋值：

```
max.i=100;
max.f=123.123;
```

执行上面两条语句后，只有 `max.f` 是有效的，而 `max.i` 已经没有任何意义了。也就是说在某一时刻只有一个成员变量起作用，其他的成员变量被覆盖，即成员变量不同时起作用。上述的赋值过程如图 9.17 所示。

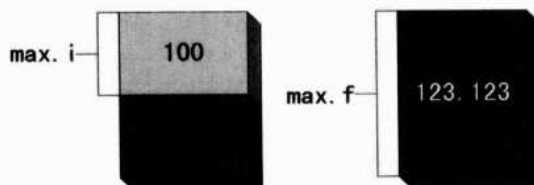


图9.17 共用体变量成员赋值

☐ 共用体变量的地址和其各个成员的地址都是同一个地址值。

这点很好理解，因为共用体变量和其各个成员共用一块内存空间，所以首地址值相同。例如，假设变量 `liming` 的首地址为 1024，那么下面变量的首地址都为 1024：

```
&liming、&liming.score、&liming.degree
```

- ❑ 不能把共用体变量用作函数参数和函数返回值，但是可以使用指向共用体变量的指针（与结构体用法相似）。

例如，下面定义共用体指针 `p_union` 并对其使用：

```
union mark *p_union;      //定义共用体指针 p_union
p_union=&liming;          //给 p_union 赋值
int i= p_union-> score;   //相当于 i=liming. score
```

- ❑ 共用体类型可以出现在结构体类型定义中，结构体类型也可以出现在共用体类型定义中；与定义结构体数组类似，也可以定义共用体数组，即如图 9.18 所示的几种方式在 C 语言中是允许的。

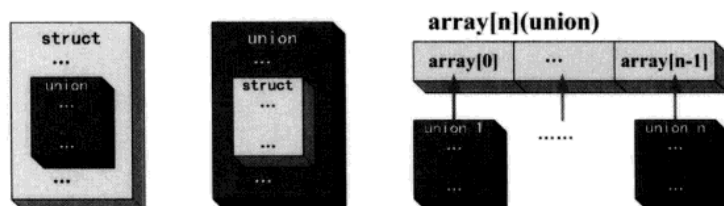


图9.18 C语言允许的共用体形式

9.2.5 结构体和共用体综合实例——“梁山好汉的比武大会”

笔者大胆地做了一个假设，住在梁山的 108 个好汉因为他们的排名问题闹起了矛盾，主要是因为有的人的技艺已经有很大的提升，而有些人占着自己靠前的排名坐享其成。经过高层领导商议之后，决定举办一场比武大会，按照比武的成绩重新确立好汉们的排名，并且根据比武结果的等级来颁发奖品。

现在分配给你一个任务，编写程序将比武结果公布于众。具体来说，需要公布好汉的原有排名，再根据比武成绩公布现有排名，以及公布获奖情况，如果是前三甲，还要公布好汉详细的个人信息。其中，分数从 1~108 分，根据分数给予奖励，“一等奖”、“二等奖”、“三等奖”和“安慰奖”分别对应 100~108、90~99、80~89、1~79。

【分析】这个任务可以分为两个大的方面，一是好汉们信息的维护，主要是修改排名，用结构体类型实现是一个很好的选择；二是对成绩和奖项的维护，用共用体类型实现较为合理。

下面给出比武信息公布的程序清单：

```
#include <stdio.h>
#include <string.h>

//结构体声明
struct hero
{
    int number;           //排名
    int new_number;       //新排名
    char star_name[20];   //星名
    char name[20];        //名字
    char nickname[20];    //绰号
```



```
char position[20]; //职业
char weapon[20]; //使用武器
};
//定义表示 108 位好汉的结构体数组 person 和表示前三甲的结构体数组 well
struct hero person[108];
struct hero well[3]={{1,-1,"天魁星","宋江","及时雨","总首领","无"),
{2,-1,"天罡星","卢俊义","玉麒麟","总督兵马副元帅","麒麟黄金矛"},
{3,-1,"天机星","吴用","智多星","机密军师","八门金锁阵"}};

//共用体声明
union mark
{
    int score; //表示分数
    char degree[10]; //表示奖项
};
//定义表示成绩和奖项的共用体 result
union mark result;

//函数声明
void change_number(int);
void init_person(int);
void init_well(int);
void pf_well(void);
void get_cheer(int);

//主函数
void main()
{
    int i;

    for(i=0;i<108;i++)
    {
        init_person(i);
        change_number(i);
        if(person[i].new_number<=3)
        {
            init_well(i);
        }
        get_cheer(i);
    }
    pf_well();
}

/*
    此函数用于改变排名,
    形参为 person 的下标,
    无返回值
*/
void change_number(int i)
{

```

```
printf("请输入%s 比赛的成绩(1-108): \n", person[i].name);
scanf("%d", &result.score); //给表示成绩的变量赋值
person[i].new_number = 109 - result.score; //计算新的成绩
printf("比武后%s 的排名是: %d\n", person[i].name, person[i].new_number);
}

/*
    此函数用于初始化结构体数组 person,
    形参为 person 的下标,
    无返回值
*/
void init_person(int i)
{
    printf("请输入第%d 位好汉的排名、星名、名字、绰号、职业、使用武器\n", i+1);
    scanf("%d%s%s%s%s",
        &person[i].number, person[i].star_name, person[i].name,
        person[i].nickname, person[i].position, person[i].weapon);
    printf("比武前%s 的排名是: %d\n", person[i].name, person[i].number);
}

/*
    此函数用于初始化结构体数组 well,
    形参为 person 的下标,
    无返回值
*/
void init_well(int i)
{
    int j;

    j = person[i].new_number - 1;
    well[j] = person[i];
}

//此函数用于输出前三名
void pf_well(void)
{
    int j;

    printf("比赛的前三甲是: \n");
    for(j=0; j<3; j++)
    {
        printf("第%d 名的信息为 (星名、名字、绰号、职业、使用武器): \n", j+1);
        puts(well[j].name);
        puts(well[j].starname);
        puts(well[j].nickname);
        puts(well[j].position);
        puts(well[j].weapon);
    }
}
```



```
/*
    此函数用于输出奖项，
    形参为 person 的下标，
    无返回值
*/
void get_cheer(int i)
{
    switch((person[i].new_number)/10)
    {
        //给共用体变量 result 的成员 degree 赋值
        case 0:
            strcpy(result.degree, "一等奖"); break;
        case 1:
            strcpy(result.degree, "二等奖"); break;
        case 2:
            strcpy(result.degree, "三等奖"); break;
        default:
            strcpy(result.degree, "安慰奖");
    }

    printf("%s 在这次比武中获得: ", person[i].name);
    puts(result.degree);
}
```

此程序综合运用了结构体和共用体，实现了假设的梁山比武大会的公布排名和奖励的功能。因程序比较简单，笔者不再进行解释，但此程序并不完善，笔者认为此程序存在以下几点问题需要读者进行改进：

- 对好汉们的信息的输入完全取决于用户，无法保证用于输入一些不合常理的值。

例如，如果用户对于好汉的成绩输入了相同的值，则好汉新的排名也将重复，这明显不合要求。

- 从整体结构上看，并没有使用第 8 章介绍过的预处理功能将程序分为若干个文件。

例如，结构体和共用体声明以及函数声明可以放在一个头文件中以供调用。

9.3 枚 举

笔者在第 2 章介绍 C 语言的数据类型时，就提到了一种基本数据类型——枚举（enum）类型，但此后这种类型就再未“现身”。之所以到此处才对枚举类型进行介绍，是因为它从外部形式上更像一种构造类型，但它确实是一种基本类型。本节就来介绍这个“表里不一”的特殊的基本类型。

9.3.1 什么是枚举

随着计算机科学的飞速发展，如今的程序已不仅仅只用于数值计算，而更广泛地应于非数值计算中。例如，如果用数值表示诸如性别、月份、星期、颜色这样的数据，既不直观，也不易读。如果能在程序中用自然语言中与其相应含义的单词来代表某一状态，则程序就很容易阅读和理解了。而诸如 int、float 类型就无法对其进行处理，从而引入了一种新

的类型——枚举类型。

在编写程序时，经常遇到变量取值能够列举的情况，例如，表示星期的变量取值有星期一到星期天 7 个值，表示月份的变量取值有一月到十二月 12 个值，表示三原色的变量取值有红、绿、蓝 3 个值等。为了更好地处理这一类变量，使程序更直观，处理更有效，ANSI C 新标准增加了枚举数据类型，将变量允许的取值明确列举了出来，变量的取值只限于列举出来的常量。之所以叫枚举是因为它将变量可能的取值一一列举了出来。

枚举类型的定义形式与结构体和共用体类似，其一般形式如下：

```
enum 枚举名{枚举值表};
```

例如，可用来表示三原色的枚举类型定义如下：

```
enum meta_color{red, green, blue};
```

这个枚举类型如图 9.19 所示。

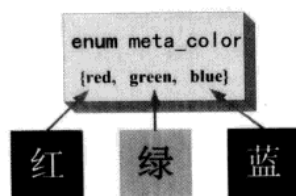


图9.19 枚举类型

9.3.2 枚举变量的定义与取值

在定义了枚举类型后，就可以定义这个枚举类型的变量。定义的一般形式与结构体和共用体类似，也分为 3 种形式。

☑ 先定义枚举类型，再定义枚举变量：

```
enum 枚举名{枚举值表};
enum 枚举名 变量名列表;
```

☑ 定义枚举类型的同时定义枚举变量：

```
enum 枚举名{枚举值表} 变量名列表;
```

☑ 不使用枚举名，直接定义枚举变量：

```
enum {枚举值表} 变量名列表;
```

下面使用 3 种不同的形式定义 3 个不同的变量：

```
enum meta_color red_ball;
enum meta_color{red, green, blue} green_ball;
enum {red, green, blue} blue_ball;
```

枚举变量只能从枚举类型列出的枚举元素中取值。例如，下面的赋值语句是正确的：

```
red_ball=red;
green_ball=green;
enum {red, green, blue} blue_ball=blue;
```

如果试图从枚举类型列出的元素外取值，则会发生错误。例如下面的赋值语句就会发生错误：



```
enum meta_color black_ball;
black_ball=black; //meta_color 中不含有元素 black
```

此处再来强调一点，编译器并不为枚举名分配内存空间，只为枚举变量分配内存空间。例如，前面定义的枚举名 `meta_color` 和枚举变量 `red_ball` 和 `green_ball`，如果把小球是否有颜色看做是否具有内存空间，那么 `meta_color` 是无色的，而 `red_ball` 和 `green_ball` 在被赋值后变得有色，如图 9.20 所示。

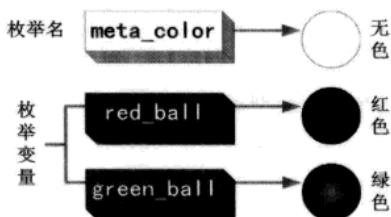


图9.20 枚举名与枚举变量

9.3.3 “表里不一”的类型

笔者反复强调枚举类型虽然从外部形式上看类似于构造类型，但实质上是一种基本类型。这是因为枚举元素实质上是常量，也可将其叫做枚举常量。枚举常量的数据类型是 `int` 型，在默认情况下枚举常量的值是在枚举值表中的顺序号。从第一个枚举元素开始，顺序号依次为 0、1、2、……。例如，表示三原色的枚举类型：

```
enum meta_color{red, green, blue};
```

实质上这个三原色的枚举类型如图 9.21 所示。

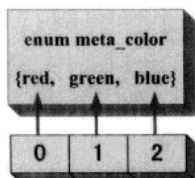


图9.21 枚举常量的默认值

当然，也可以在定义枚举类型时，给枚举常量指定值，其一般形式如下：

```
enum 枚举名{
    枚举常量 1=整数常量值,
    枚举常量 2=整数常量值,
    ...
    枚举常量 n=整数常量值
};
```

例如，下面是对表示三原色的枚举类型常量指定值：

```
enum meta_color{red=10, green=20, blue=30};
```

如果只对一个枚举常量赋值，而没有给后面的枚举常量赋值，那么后面的常量会被赋予后续的值。例如：

```
enum meta_color{red, green=20, blue};
```

在这个声明中，枚举常量 `green` 进行了赋值，其值为 20，而 `red` 取默认值为 0，`blue` 取 20 的下一个值为 21。给枚举常量全部赋值和部分赋值如图 9.22 所示。

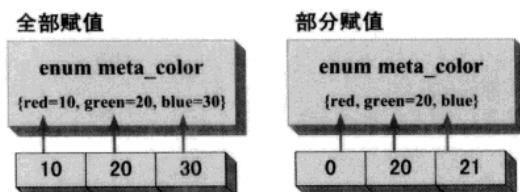


图9.22 枚举常量的全部赋值和部分赋值

既然枚举常量是一种 `int` 型的常量，那么当然可以使用在 `int` 型常量的任何地方。例如，下面对枚举常量的使用都是正确的：

```
enum weekday{Sun,Mon,Tue,Wed,Thu,Fri,Sat}; //声明枚举类型
int array[Sat+1];                          //枚举常量做数组大小
switch(Fri)                                //枚举常量做 switch 的标签
{
    ...
}
```

虽然枚举常量的值被限定为 `int` 型，但是枚举变量的类型较为宽松，只被限定为任何一种整数类型，要求该类型能够保存枚举常量即可。例如，下面的枚举变量 `ball` 可以是 `char` 型，因为其枚举常量的值为 0~2：

```
enum {red, green, blue} ball;
```

9.3.4 枚举应用举例——“向你问好的程序”

下面的程序完成这样的功能：根据用户输入的时间段（0 表示上午，1 表示下午，2 表示晚上）来输出相应时间的问好。将其称为“问好程序”，程序清单如下：

```
#include <stdio.h>

//枚举声明
enum time_of_day
{
    Morning,
    Afternoon,
    Evening
};

//函数声明
void hello(enum time_of_day);

void main()
{
    enum time_of_day tod;

    puts("请输入时间段（0 表示上午，1 表示下午，2 表示晚上）");
```



```
scanf("%d",&tod);

hello(tod);
}

/*
    根据输入，输出不同的问候语，
    形参为枚举类型变量，
    无返回值
*/
void hello(enum time_of_day tod)
{
    switch(tod)
    {
        case Morning:
            puts("Good morning!");break;
        case Afternoon:
            puts("Good afternoon!");break;
        case Evening:
            puts("Good evening!");break;
        default:
            puts("Hello!");
    }
}
```

此程序充分说明了枚举常量和变量的实质。因为枚举值是一个常量，所以定义的枚举值 Morning、Afternoon、Evening 完全可以像 0、1、2 这样的整型常量一样运算；并且枚举变量 tod 也可以用 %d 的形式进行输入，并且可以作为 switch 语句的标签，可见枚举类型确实是一个整数类型。

9.4 用户自定义类型——typedef

对于数据类型，C 语言可算是做到“仁至义尽”，它提供的内置数据类型完全可以满足程序开发时的各种需要。但是在个别时候，内置的数据类型可能不是开发程序时数据的最好表示。强大的 C 语言当然不会“坐视不管”，它还提供了一种可以由用户来自己定义数据类型的方式——typedef。

9.4.1 什么是 typedef

typedef 被称为用户自定义类型。但其实并不是用户自己定义的一种新的数据类型，而是用户根据自己的具体需要给某种数据类型的重新命名，新的命名可以叫做别名。其定义语句一般形式为：

```
typedef 类型名 标识名;
```

其中，“类型名”必须是在此之前已有定义的类型标识符。“标识名”是一个用户定义的标识符，标识新的类型名。其作用为声明一个用户自定义的类型，数据类型为“类型名”声明的类型。例如，下面定义一个表示点的数据类型：




```
struct pts
{
    int x;
    int y;
};
typedef struct pts Point;
```

该语句把一个用户命名的标识符 Point 说明成一个 struct pts 类型的新类型名。定义后, 就可以用 Point 定义变量了。例如:

```
Point start,end; //定义变量 start, end 为 Point 型的变量
```

等价于:

```
struct pts start,end;
```

 **注意:** typedef 实质上是对现有数据类型的另一种说明, 并没有增加新的数据类型。即标识名只是类型名的一个别名, 并不真正存在以标识命名的类型。

9.4.2 创建 typedef 简单方法

使用 typedef 定义类型时, 类型名可能会非常复杂, 必须要有一种好的方法来定义满足自己需要的类型名的变量。此处介绍一种简单有效的方法。

第 1 步 首先按照一般定义变量的方法写出定义语句。

第 2 步 将变量名换为新类型名。

第 3 步 在最左边加上关键字 typedef。

第 4 步 用新类型名定义变量。

例如, 定义一个表示字符指针类型的新类型 PCHAR, 按照上述方面可分为如下四步:

第 1 步 char *p;

第 2 步 char *PCHAR;

第 3 步 typedef char *PCHAR;

第 4 步 PCHAR x,y;

同样, 下面使用上述方法定义一个包含 100 个元素的整形数组类型, 其别名为 HARRAY:

第 1 步 int a[100];

第 2 步 int HARRAY[100];

第 3 步 typedef int HARRAY[100];

第 4 步 HARRAY arr;

9.4.3 typedef 和#define

使用 typedef 和#define 指令都可以定义自己命名的数据类型。不过两者之间仍然具有以下几点区别:

☑ typedef 只针对数据类型, 不针对值, #define 则既可以针对数据类型, 也可以针对值。



例如，下面使用#define 指令对变量 x 进行强制类型转换是正确的：

```
#define TO_INT(x)  int(x)
```

但如果换成 typedef 就会发生错误：

```
typedef TO_INT(x)  int(x);
```

☐ typedef 是一种对已有数据的类型的彻底“封装”，在声明之后不能在往其中增添任何代码。而 define 只是简单的文本替换，定义之后仍然可以增加一些合法的代码。

例如，下面使用 typedef 的代码是非法的：

```
typedef int H_NO;  
unsigned  H_NO  h; //非法的定义
```

但是对于#define 指令则没有问题：

```
#define int H_NO  
unsigned  H_NO  h; //合法的定义
```

☐ 在定义几个连续的变量的声明中，用 typedef 定义的变量可以保证所有的变量都为同一数据类型。但是用#define 定义的类型则无法保证。

例如，下面定义的变量 p1、p2 都是*char 类型的：

```
typedef char  *PCHAR;  
PCHAR  p1,p2; // p1、p2 都为字符指针类型
```

而下面的变量 p1 是 char *型，p2 则是 char 型：

```
#define  PCHAR  char *  
PCHAR  p1,p2;
```

因为在进行宏替换后语句变为：

```
char *p1,p2;
```

9.4.4 typedef 的两个重要作用

1. 使用 typedef 有利于程序的移植和跨平台开发

typedef 有一个重要的用途，那就是定义与机器无关的类型。例如，前面介绍的运算符 sizeof 的数据类型为 size_t 类型，是一种自定义类型。ANSI C 只规定了 sizeof 的值为整数类型，而它是哪种整数类型留给了具体实现。

标准 C 委员会觉得对所有计算机平台来说，没有哪一种类型是最佳的选择。因此给出了一种新的类型名 size_t，让具体的实现者使用 typedef 来将这个类型名定义为特定的类型。size_t 的定义包含在头文件 time.h 中，许多机器上都将其定义为 unsigned int 类型：

```
typedef unsigned int  size_t;
```

2. 使用 typedef 可以简化代码

简化复杂的代码是 typedef 的另一个重要的应用，假设程序中要经常定义结构体指针，就可以使用 typedef 定义一个类型如下：

```
typedef struct hero *PHero;  
PHero plinchong;  
PHero pluda;
```



这样的代码不但简洁，而且不易出错，便于理解。同样，可以使用 `typedef` 简化复杂的函数声明。例如，下面是一个复杂的函数指针的声明：

```
int (*pfun)(int, int);
```

如果程序中要经常使用这种类型的声明，就可以使用 `typedef` 进行简化：

```
typedef int (*function_p)(int, int);
```

这样就可以定义多个这种函数指针，例如，下面定义 3 个这样的函数指针：

```
function_p fun1;  
function_p fun2;  
function_p fun3
```

当然，也可以在定义时对其进行初始化，假设函数 `fun_1()`、`fun_2()`、`fun_3()` 是已经定义的 3 个函数，则可以用下面的形式对这 3 个函数指针进行初始化：

```
function_p fun1 = fun_1;  
function_p fun2 = fun_2;  
function_p fun3 = fun_3
```





第 10 章

文 件



计算机中的程序、数据等都是以文件的形式来保存的。例如一张照片可能以 BMP 或者 JPG 的形式来保存，一份文档可能以 DOC 的形式来保存，一首歌曲可能以 MP3 或者 WAV 的形式来保存，这里所说的 BMP、JPG、DOC、MP3 和 WAV 就是所谓的文件类型。文件是计算机在外部存储器上（例如硬盘、光盘等）记录数据集合的形式。尽管从计算机用户的角度来看好像文件分为很多种（例如 BMP、JPG 或者 DOC 等），但是在处理这些文件时，它们本质上就只有两种形式，即文本文件和二进制文件。本章就来研究利用 C 语言处理文本文件和二进制文件的方法。



10.1 理解文件的基本概念

在研究如何处理文件之前,首先向读者介绍一些和文件有关的基本概念,它们将帮助读者更好地学习 C 语言中文件处理的基本方法。

10.1.1 什么是文件

1. 文件定义

“文件 (file)”就是指存储在外部存储器上的数据集合。更准确地说,文件就是一组相关元素或数据的有序集合,而且每个集合都有一个符号化的指代,我们称这个符号化的指代为“文件名”。

文件通常都被存在外部存储器上,它只有在需要使用时才会被调入内存。实际上在前面的各章中已经多次使用了文件,例如源程序文件、目标文件、可执行文件、头文件,以及我们所列举的 BMP 文件、JPG 文件和 DOC 文件等都是常见的文件实例。

2. 普通文件和设备文件

从更加广义的角度来说,文件还可分为普通文件和设备文件两大类,如图 10.1 所示。

- ☐ 普通文件:指驻留在磁盘或其他外部介质上的一个有序数据集,因此前面所列举的各种文件都属于此类。
- ☐ 设备文件:主要是指与主机相连的各种外部设备,如显示器、打印机、键盘等。

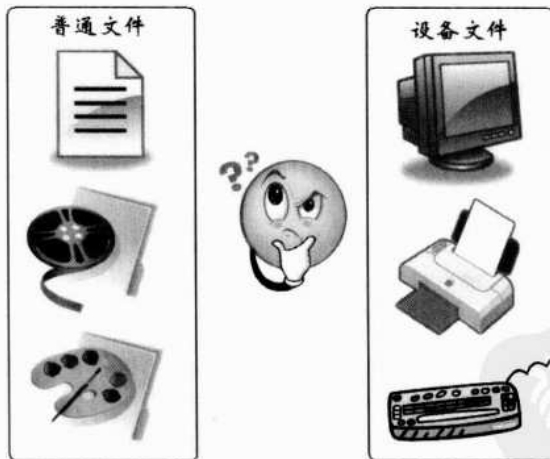


图10.1 普通文件和设备文件



注意:操作系统把外部设备也看作是一个文件来管理,与普通的数据文件不同,它们只是一种逻辑上的文件。

在通常情况下,显示器被定义为标准输出文件,键盘则被定义为标准输入文件。因此,在屏幕上显示信息就是向标准输出文件输出信息,如前面经常使用的 `printf()`、`putchar()` 等函



数就属于这类输出；从键盘上键入数据就意味着从标准输入文件上接收数据，如 `scanf()`、`getchar()` 等函数就属于这类输入。

文件概念的引入使所有的系统资源（普通文件或目录、磁盘设备、键盘、显示器、打印机等）有了统一的标识，操作系统对这些资源的访问和处理都是通过统一的字节序列的方式来实现的。



注意：前面所研究的标准输入/输出主要是面向设备文件的，本章所研究的文件主要是指除了设备文件外的普通文件。如无特殊说明本章后面所出现的文件即专指普通文件。

10.1.2 什么是流

1. 流的概念

流（Stream）是一种逻辑上的概念。在 C 语言中，把任何输入/输出的数据都视为“流”，流就是程序输入/输出的一个连续数据序列。读者已经知道，所有文本输入和输出数据实际上都是字符代码的连续流，所以有时将源数据或目的数据称为输入流或输出流。

系统为了简便操作，将普通文件和设备文件等同看待，因此输入流或输出流对于普通文件和设备文件都是通用的。所以流就是表示文本输入（或输出）数据的字符代码的连续序列。

2. 流文件

在 C 语言中，流作为连续的数据序列不是由记录组成的，所以文件输入/输出的字节流或二进制流仅受程序控制而不受物理符号（如回车换行符）控制。也就是说，文件在输入/输出时不会考虑记录的界限，这种文件通常可以称为流文件。



提示：在 C 语言中，所有的流均以文件的形式出现。

从另一个角度来说，流实际上是文件输入/输出的一种动态形式。在 C 语言中“普通文件”就是一个字节流或二进制流。

3. 流的工作原理

流在启动时，会自动创建 3 个标准文件指针 `stdin`、`stdout` 和 `stderr`，这 3 个指针分别指向和控制 3 个预定义的“标准文件流”。这 3 个“标准文件流”分别是：

- ☑ 从操作系统向标准输入设备（通常为键盘）输入数据所构成的“标准输入流”。
- ☑ 从操作系统向输出设备（通常为屏幕）输出信息所构成的“标准输出流”。
- ☑ 由屏幕显示报告出错信息构成的“标准报错流”。

此外，C 语言还提供了其他两个流：

- ☑ 标准打印机。
- ☑ 标准串行设备。

这 5 个标准流可以被应用在程序的任何地方，而且不必再被打开或关闭。表 10.1 列出了这 5 种标准流。

表10.1 文件中的标准流

名 称	描 述	例 子
stdin	标准输入	键盘
stdout	标准输出	屏幕
stderr	标准错误	屏幕
stdprn	标准打印机	LPT1 端口
stdaux	标准串行设备	COM1 端口

简单地说,流是磁盘或其他外围设备中存储数据的源点或终点。产生数据的叫做输入流,消耗数据的叫做输出流,为了消除各种设备之间存在的差异性,C语言对每种设备都一视同仁,以一个“流”字来概括它们的特征。因此作为“流”的使用者,不需要关心太多的实现细节,只需要掌握如何使用流来完成对文件的各种操作即可。

10.1.3 处理文件的方法

在C语言中有两种处理文件的方法:

- ☑ 缓冲文件系统(有时也叫格式化系统、文本系统或高级系统)。
- ☑ 非缓冲文件系统(有时也叫非格式化系统或二进制系统)。

“缓冲文件系统”是指系统自动在内存区域为每个正在使用的文件开辟一个缓冲区,从内存向外部介质输出的数据必须先送到缓冲区,待缓冲区装满后才送到外部介质;在读入数据时,依次从外部介质中将一批数据输入到缓冲区,然后依次从缓冲区将数据送到程序数据区,赋值给程序中的变量。缓冲文件系统对数据的操作流程如图10.2所示,缓冲区的大小由具体的C版本确定,一般为512KB。

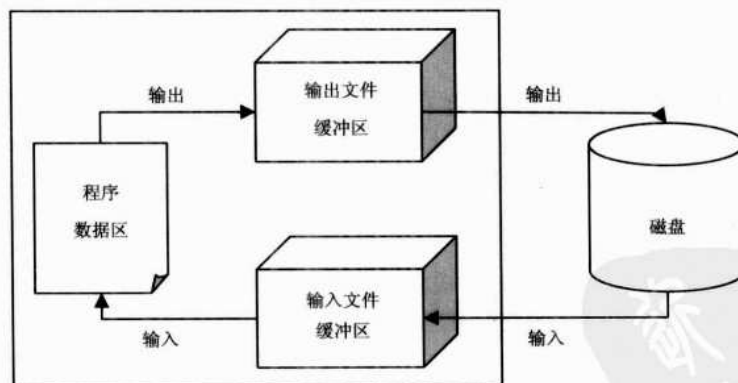



图10.2 缓冲文件系统

“非缓冲文件系统”是指系统不会自动为文件开辟确定的缓冲区,而由程序为每个文件设定缓冲区,但目前的ANSI C不提倡使用“非缓冲文件”。



 **注意：**在某些操作系统（如 UNIX）中，文本文件和二进制文件的处理采用的是不同的文件处理方法，而 ANSI C 统一规定使用缓冲文件系统，利用缓冲文件系统来完成对文本文件和二进制文件的读/写操作。本章所介绍的文件操作只是针对 ANSI C 中规定的缓冲文件系统而言的。

10.1.4 缓存

1. 文件缓存过程

由于每一次系统对文件的 I/O 调用都很费时间，因此标准 I/O 提供缓存的目的就是为了减少文件系统 I/O 次数，对每个流自动进行缓存管理。当程序中有数据被读入时，数据先被整块地读入缓存中，然后送到程序的数据区，这样，如果下次用户再次读取数据时，且数据正好在缓存中，就可以直接从缓存中获取，具体过程如图 10.3 所示。



图10.3 文件缓存过程


简单地说，缓存就是系统为文件在内存中开辟的一块区域，作用相当于中转站。当数据从内存向磁盘输出时，数据必须先送到缓存中，等到缓存满了或者程序请求清空缓存的时候，数据才被写入磁盘。

2. I/O 提供的缓存类型

为了方便使用，标准 I/O 提供了 3 种类型的缓存。

☐ 全缓存：

该类缓存中，文件的实际操作是在缓存被充满之后才进行的。在缓存没有完全被充满时，系统不会自动将文件写入磁盘，除非系统调用刷新（flush）操作强制将数据写入磁盘。

 **注意：**在 UNIX 环境下，刷新有两种意思，在标准 I/O 库方面，刷新意味着将缓存中的内容写到磁盘上；在终端驱动程序方面，刷新表示丢弃已存在缓存中的数据。

对于磁盘文件，通常由标准 I/O 库实施全缓存，在一个流执行第一次 I/O 操作时，标准 I/O 函数通常调用 malloc() 函数来获得所使用的缓存，此过程一般不需要人工干预。

☐ 行缓存：

该类缓存操作是在当输入和输出过程中遇到行结束标志的时候，标准 I/O 库就执行实际的 I/O 操作。例如，程序从键盘读取一个字符的时候，只有用户输入完一整行之后，程序才能够得到一个字符。通常在读取键盘和文本文件的时候，使用行缓存。

行缓存有两个限制：

- 由于系统为行缓存分配的空间有限，一般不会很大，标准 I/O 库用来收集每一行缓存的长度是固定的，所以只要填满了缓存，即使还没有写一个新字符，也进行 I/O 操作。

- 任何时候只要通过标准输入/输出库要求从一个不带缓存的流或者一个行缓存的流得到输入数据，都会造成刷新所有行缓存输出流。

☑ 无缓存：

该类缓存操作意味着系统不会为流分配缓存，每一次标准 I/O 函数的调用都会使得底层的 I/O 调用被执行，因此系统与外部介质会发生经常性的数据交换，数据操作比较慢，效率较低。标准出错流 `stderr` 通常是不带缓存的，这就使出错信息可以尽快显示出来。

10.1.5 文本文件和二进制文件

根据数据的不同组织形式，文件可分为文本文件和二进制文件两种，下面是两种文件的区别：

- ☑ 文本文件是在外部存储器中保存的一个已命名的字符结合，它又称为 ASCII 文件。而二进制文件则是指包含二进制数的文件，这些二进制数是文件中各个数据项在计算机内部的表示形式。
- ☑ 文本文件中的每一个字节中存放了一个 ASCII 代码，代表一个字符；而二进制文件则是直接用数据的二进制形式存放。例如，一个整数 123，若用 ASCII 文件存放，需占 3 个字节的存储单元，1、2、3 各用一个字节来存储；而用二进制文件存放时，只需 2 个字节，如图 10.4 所示。
- ☑ 使用文本文件，一个字节代表一个字符，便于对字符一一处理和输出，但占用较多的存储空间，并且要花费转换时间（ASCII 码与二进制之间的转换）。使用二进制文件，在内存中的数据形式与输出到外部文件中的数据形式完全一致，可以克服 ASCII 文件的缺点，但不直观，一个字节并不对应一个字符或一个数，不能直接输出字符形式。

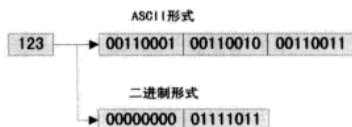



图10.4 文本文件和二进制文件存储

 注意：在 C 语言中处理文本文件和二进制文件的方法是有区别的。

10.2 文件的打开与关闭

C 语言中文件的操作是基于指针来实现的，这一点对于很多初学者来说都不太容易理解。其实，并不一定非要将文件和指针挂钩，读者只要把文件想象成一种新的数据类型就容易理解得多。和其他普通数据类型不同，文件类型数据在使用前需要“打开”，而在使用完毕后则需要“关闭”，本节就介绍如何“打开”与“关闭”文件。

10.2.1 文件类型指针

在使用 C 语言处理文件的过程中，“文件指针”是一个核心的关键概念。这涉及缓冲文件系统中如何标识每个不同文件的问题。当系统操作一个文件时，会在内存中为该文件分配一个 `FILE` 结构的内存区域，用来存放与该文件相关的信息（如文件名称、文件状态以及文件当前位置等），`FILE` 指针（亦称文件类型指针）就是指向这个 `FILE` 结构内存区域的指针变量。



在 `stdio.h` 中对 `FILE` 的结构体类型声明如下：

```
typedef struct
{
    short level;           //记录打开文件流的缓冲区填入数据的情况
    unsigned flags;        //文件状态标志
    char fd;               //文件句柄，也就是文件描述符
    unsigned char hold;    //若无缓冲区，则不读取字符
    short bsize;           //文件缓冲区大小
    unsigned char *buffer; //文件缓冲区指针
    unsigned char *curp;   //当前激活的文件指针
    unsigned istemp;       //临时文件标识
    short token;           //用于有效性检查
}FILE;
```

在 C 语言中，`FILE` 结构对文件来说非常重要，它可以用于定义 `FILE` 指针变量，不同的 C 编译器可能使用不同的定义，但基本含义变化不大。`FILE` 指针是为了对不同的文件进行区分而为每个文件赋予的一个“身份 ID”，或者可以认为 `FILE` 指针就是一个起着标记文件身份作用的指针。

以上 `FILE` 结构中的各个参数已经附有简单注释，需要读者注意的是其中的 `flags` 参数，它是一个 10 位的标志字，其具体含义如表 10.2 所示。

表10.2 FILE结构体中flags的含义

位	代表符号	含 义
0	<code>_F_READ</code>	读
1	<code>_F_WRITE</code>	写
2	<code>_F_BUF</code>	由 <code>fclose</code> 释放缓冲区
3	<code>_F_LBUF</code>	行缓冲
4	<code>_F_ERR</code>	出错标志
5	<code>_F_EOF</code>	EOF 文件尾标志
6	<code>_F_BIN</code>	二进制方式
7	<code>_F_IN</code>	进行输入
8	<code>_F_OUT</code>	进行输出
9	<code>_F_TERM</code>	文件是一个终端

有了 `FILE` 结构类型之后，可以用它来定义若干个 `FILE` 类型的变量或者指针，用来存放文件信息。例如：

```
FILE file;
```

本例中指明 `file` 是一个 `FILE` 结构体的变量。

再来看一个例子：

```
FILE *pfile;
```

本例中的 `pfile` 是一个指向 `FILE` 类型结构体的指针变量，通常被称为 `FILE` 指针。需要提醒读者注意的是不应该把文件位置指针和 `FILE` 结构指针混为一谈。它们代表两个不

同的地址。文件位置指针是文件当前的读/写位置，而 FILE 结构体指针是文件在内存中的地址。

10.2.2 文件的打开

在 C 语言中，文件打开一般采用 fopen() 函数来完成，fopen() 函数的调用形式为：

```
fopen(filename, mode);
```

该函数的返回值是一个 FILE 类型的指针变量，filename 是被打开文件的文件名，mode 定义了文件的类型和操作要求。一般来说，filename 是一个字符串常量或者字符串数组，例如：

```
FILE *pfile;
pfile = fopen("file1", "r");
```

在上述程序中，第 1 条语句定义了一个 FILE 结构体指针 pfile，第 2 条语句建立 pfile 与文件“file1”的关联，即 pfile 指向 file1 文件。从上述例子中可以看出，打开一个文件时，用户向系统提供了 3 个信息：

- ☑ 准备访问的文件名。
- ☑ 使用的文件方式（“读”、“写”还是“追加”等）。
- ☑ 用哪个指针变量指向被打开的文件，例如：

```
FILE *pfile;
pfile = ("d:\\file2.dat", "rb");
```

其含义是以只读二进制文件的方式打开位于 D 盘根目录下的文件 file2.dat，两个反斜线“\\”中的第一个表示转义字符，第二个表示根目录。如果 fopen() 函数不能完成打开指定的文件，它将返回一个空指针值 NULL。出现这种情况的原因可能是：

- ☑ 用“r”方式打开不存在的文件。
- ☑ 磁盘产生故障。
- ☑ 磁盘已满无法建立新文件。

因此，在调用打开文件函数 fopen() 之后，通常使用 if (pfile == NULL) 等语句来判断操作是否成功。

10.2.3 文件操作类型及应用

1. 文件的操作和使用方式

文件的操作类型共有 12 种，表 10.3 给出了每种类型的具体符号和意义。

表10.3 文件的操作和使用方式

文件操作类型	意 义
rt 或 r	以只读方式打开一个文本文件，只允许读数据
wt 或 w	以只写方式打开或建立一个文本文件，只允许写数据
at 或 a	以追加方式打开一个文本文件，并在文件末尾写数据
rb	以只读方式打开一个二进制文件，只允许读数据
wb	以只写方式打开或建立一个二进制文件，只允许写数据



(续表)

文件操作类型	意 义
ab	以追加方式打开一个二进制文件，并在文件末尾写数据
rt+或 r+	以读/写方式打开一个文本文件，允许读和写
wt+或 w+	以读/写方式打开或建立一个文本文件，允许读和写
at+或 a+	以读/写方式打开一个文本文件，允许读，或在文件末追加数据
rb+	以读/写方式打开一个二进制文件，允许读和写
wb+	以读/写方式打开或建立一个二进制文件，允许读和写
ab+	以读/写方式打开一个二进制文件，允许读，或在文件末追加数据

2. 6 个字符的含义

文件使用方式由 r、w、a、t、b 和+6 个字符组成，各字符的含义是：

- ☑ r (read)：读。
- ☑ w (write)：写。
- ☑ a (append)：追加。
- ☑ t (text)：文本文件，可省略不写。
- ☑ b (binary)：二进制文件。
- ☑ +：读和写。

用“r”方式打开一个文件时，该文件必须已经存在，且只能从该文件读出数据，不能用此方式打开一个不存在的文件，否则出错。用“w”方式打开的文件只能被写入数据。若指定打开的文件不存在，则以该文件名新建一个文件，若打开的文件已经存在，则将原文件覆盖。用“a”方式对文件追加新的信息，但在追加之前必须保证该文件存在，否则会出错。

用“r+”、“w+”、“a+”方式打开的文件可以用来输入数据，也可以用来输出数据。用“r+”方式打开文件时该文件必须已经存在；用“w+”方式打开文件时，如果文件不存在将会新建一个文件，然后对该文件进行读/写操作；用“a+”方式打开文件时，原来的文件将不会被删除，并且文件位置指针自动移动到文件末尾，然后对该文件读/写。

3. 文本文件和二进制文件的读/写操作

操作系统对文本文件和二进制文件的读/写操作是不同的。文本文件在从内存写入磁盘时，需要将二进制码转换成 ASCII 码；文本文件在从磁盘读入内存时，也需将 ASCII 码转换成二进制码，因此文本文件的读/写操作需花费较多的转换时间。对二进制文件的读/写不存在这种转换。

在向计算机输入文本文件时，将回车换行符转换为一个换行符，在输出时把换行符转换为回车和换行两个字符；而在二进制文件操作时，不需要进行这种转换，内存数据与磁盘中的数据形式完全一致，并一一对应。

ANSI C 规定，以上方式可以打开文本文件或者二进制文件，并用同一种缓冲文件系统来处理，但需要注意的是并不是所有的 C 编译系统都提供这些功能，例如某些编译系统只提供“r”、“w”、“a”方式，因此用户在使用之前必须注意所用系统的规定。

4. 文件出错处理

在打开一个文件时，如果出错，`fopen()`将返回一个空指针值 `NULL`。因此，在程序中可以用返回值是否为空指针来判断是否成功完成打开文件的操作，并进行相应的处理。例如程序中常用下面形式打开文件：

```
FILE *pfile;
//打开文件并判断，若不成功则返回空值
if((pfile=fopen("c:\\file2.dat", "r"))==NULL)
{
    printf("\n Error on open c:\\file2.dat!");
    getch();
    return 0;
}
```

在上面程序中，第3行打开文件操作中如果返回一个空指针，表示不能打开C盘根目录下的 `file2.dat` 文件。这时，在换行后给出提示信息“Error on open c:\\file2.dat!”，第6行 `getch` 是从键盘上输入一个字符，这里的作用是等待，允许用户有时间查看出错信息，当用户从键盘上任意输入一个字符时，程序继续运行。第7行“`return 0;`”语句终止调用的过程，退出本程序。

11.2.4 文件的关闭

文件在使用完之后，必须将其关闭，这是因为打开的文件在进行写入操作时，若文件缓冲区的空间未被填满，这些数据不会被实际写到文件中去，产生数据丢失。打开的文件只有在关闭时，停留在文件缓冲区的内容才能被写入实际的磁盘文件中，同时关闭文件时可以释放与其对应的 `FILE` 结构内存。

如果总是执行打开文件操作，而不进行关闭，那么当未关闭的文件逐渐增多时，系统的内存资源会被消耗殆尽，这就跟使用 `malloc` 分配内存资源必须用 `free` 来显式地释放一样，这样才能确保系统有限的内存资源得到合理利用。

文件关闭函数 `fclose()` 的调用形式为：

```
fclose(pfile);
```

其中，`pfile` 是一个 `FILE` 类型指针，指向一个已经被打开的文件。`fclose()` 在关闭文件前先清除文件缓冲区，若关闭成功则返回 0，否则返回 `EOF (-1)`。通常用以下方法对文件进行关闭：

```
if ( fclose(pfile) != 0 )           //关闭 pfile 指向的文件，并判断是否成功
{
    printf("File cannot be closed\n"); //若关闭文件失败，打印出错误信息
    return 0;                          //退出程序
}
else
    printf("File is now closed!\n");   //若程序关闭，则提示已经成功关闭
```

在上述程序段中，程序关闭 `pfile` 指向的文件，若操作成功则显示“File is now closed”，否则显示“File cannot be closed”并退出。

当对打开的多个文件同时进行关闭时，可以使用 `fcloseall()` 函数，它将关闭所有在程序



中已经打开的文件。它的调用形式为：

```
fcloseall();
```

该函数将关闭所有已打开的文件，并将各个文件缓冲区未装满的内容写入相应的文件中，然后释放掉这些缓冲区，并返回关闭文件的数目。例如关闭 5 个文件，则当执行以下程序时，其返回的值 *n* 应该是 5。

```
int n;  
n = fcloseall();
```

下面给出一个完整的示例程序用以说明文件的打开与关闭操作。该程序的作用是输入指定文件中的一个字符串，求出该字符串中包含字母“a”的个数，并将其打印出来。为了能获得文件中的字符串，必须先打开该文件，读取其中的数据，然后将求得的结果输出到屏幕上，最后关闭文件。根据以上的分析，实现代码如下：

```
#include <stdio.h>  
#include <iostream.h>  
  
int main()  
{  
    int num=0,i=0;  
    char str[255];  
  
    FILE *pfile;  
  
    if ((pfile=fopen("D:\\\\file.dat","r"))==NULL )  
    {  
        printf("Error on open file!\\n");  
        getchar(); //等待用户输入任意字符继续  
        return 0; //退出程序  
    }  
    else  
    {  
        fgets(str, 255, pfile);  
        for ( i = 0; i<=255; i++)  
        {  
            if (str[i]=='a')  
                num++;  
        }  
        printf("该文件中含\\\"a\\\"字母的个数为:%d\\n",num);  
        fclose(pfile); //关闭 pfile 指向的文件  
    }  
  
    return 1;  
}
```

请读者完成编码后编译并运行程序，并观察输出结果。



注意：在 C 语言中，对于文件的操作分为几个部分：文件打开→文件数据读取→写入→文件关闭。文件的打开和关闭是对文件操作的前提，对文件数据的读/写是文件操作的核心部分。

10.3 文件的基本操作

在文件打开以后,就可以对文件进行读/写操作。本节就介绍在 C 语言中对文件进行读/写操作的基本方法。

10.3.1 文件中的字符读/写

从文件中读取一个字符或者向文件中写入一个字符是对文件操作的一种最简单的形式,在 C 语言中,这些操作是通过 `fputc()`、`fgetc()` 等函数实现的。

1. `fputc()` 函数

函数 `fputc()` 的功能是把一个字符写入磁盘文件,它的调用形式一般为:

```
fputc( c, pfile );
```

其中 `c` 是要输出的字符,可以是字符常量或者变量, `pfile` 是已定义过的文件指针,该函数把一个字节的字符值 `c` 写入 `pfile` 指向的文件中,并返回一个值用来表示操作是否成功的值,如果操作成功该函数返回要写入的字符,否则返回 `EOF` (`-1`), `EOF` 是 `stdio.h` 中定义的一个符号常量,其值为 `-1`。

下面给出使用 `fputc()` 函数的一个示例:

```
fputc( 'a', pfile );
```

上述例子的含义是把字符“a”写入 `pfile` 所指向的文件中。

在使用 `fputc()` 函数时,应该注意以下几点:

- ☑ 被写入的文件可以用写、读/写、追加方式打开,用写或读/写方式打开一个已存在的文件时将清除原有的文件内容,写入的字符从文件首开始。如需保留原有文件内容,希望写入的字符以文件末开始存放,必须以追加方式打开文件。被写入的文件若不存在,则创建该文件。
- ☑ 每写入一个字符,文件位置指针向后移动一个字节。
- ☑ `fputc()` 函数有一个返回值,如写入成功则返回写入的字符,否则返回 `EOF`,可用此来判断写入是否成功。

2. `fgetc()` 函数

函数 `fgetc()` 的功能是从指定的文件中读取一个字符,函数的调用形式为:

```
fgetc(pfile);
```

其中 `pfile` 是一个 `FILE` 结构体指针,指向已经打开的一个文件。若操作执行成功,则函数返回值是一个读取出的字符,若当时文件位置指针已经达到文件尾(即遇到文件结束标志(`EOF`)),则返回 `-1`。因此程序中经常用该函数的返回值是否为 `EOF` (`-1`) 来判断是否达到了文件的末尾,从而决定是否继续读取。例如:

```
char ch;
ch = fgetc(pfile);           //从 pfile 指向的文件中获取一个字符,传给字符变量 ch
while(ch != EOF)             //判断是否到达文件末尾
{
    putchar(ch);              //如果没有达到文件末尾则将读取的字符打印在屏幕上
}
```



```
ch = fgetc(pfile);    //继续获得下一个字符  
}
```

本程序段的功能是从文件中逐个读取字符，并显示到屏幕上。`pfile` 是程序中已经定义过的 `FILE` 类型指针，而且已经指向一个打开的文件。程序首先读出一个字符，然后进入循环，只要读出的字符不是文件结束标志 (EOF)，程序就把该字符显示在屏幕上，再读入下一字符，此过程如图 10.5 所示。每读一次，文件位置指针向后移动一个字符，当文件结束时，该指针指向 EOF。本程序段会将文件中的内容逐个显示到屏幕上。

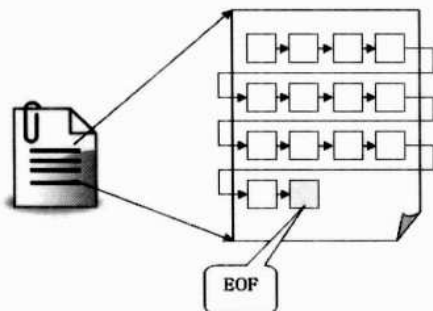



图10.5 逐个读取文件中的字符

用户在使用 `fgetc()` 函数时应该注意以下几点：

- ☑ 在 `fgetc()` 函数调用中，读取的文件必须是以读或读/写方式打开的。
- ☑ 读取字符的结果也可以不向字符变量赋值。但在这种情况下，读出的字符将不能被保存，例如：

```
fgetc(pfile);
```


- ☑ 文件内部在进行读/写操作时，文件位置指针是移动的。例如在打开文件时，该指针总是指向文件的第一个字节。在使用 `fgetc()` 函数后，该位置指针往后移动一个字节。因此，在对文件进行读/写操作时，可以使用 `fgetc()` 函数连续多次读取字符。

 **注意：** `FILE` 指针和文件位置指针的概念，并对其仔细区分。`FILE` 指针是指向整个文件的，需在程序中定义说明，只要不重新赋值，`FILE` 指针的值就不会变；文件位置指针用以指示文件内部的当前读/写位置，每读/写一次，该指针向后移动一个字节，它不需要在程序中定义说明，而是由系统自动设置的。

- ☑ 在对二进制文件进行处理时，由于某一个读入的字符值很可能正好是 -1，C 语言编译器就有可能将这个与文件结束标志 EOF 混淆，从而引起误操作，将需读取的字符误当做是“文件结束”。因此在对二进制文件操作时，可以使用 `feof()` 函数来判断文件当前状态是否为“文件结束”。`feof()` 的调用形式如下：

```
feof( pfile );
```

在文件结束时，该函数返回值为 1，否则返回 0。

 **注意：** 这种方法也适用于文本文件。

10.3.2 按字符进行读/写文件——文件复制的功能

文件备份是保证数据安全的一种有效措施。操作系统中已经提供了文件复制的命令，在 Windows 中，文件复制操作只要通过右键菜单中的“复制”命令即可轻松完成。那么读者有没有想过系统是如何实现该操作的呢？在此我们使用本节前面所学习的一些 C 语言命令来实现这个文件复制的功能。

这个程序首先接受用户输入的源文件名，并打开文件，然后打开目标文件。如果这两个文件都顺利打开，则从源文件中逐个读取字符并复制到目标文件中，操作完成后在屏幕上输出操作成功的信息。下面给出该程序的实现代码清单：

```
/*
 * 实现备份文件功能
 * 运用 getc() 和 putc() 函数实践
 */
#include <stdio.h>
#define SIZE 80

int main()
{
    char source_file[SIZE]; // 源文件名称
    char dest_file[SIZE];  // 目标文件名

    FILE * inFile, * outFile;
    char ch;

    // 打开源文件
    printf("请输入欲备份的文件名称> ");
    for(scanf("%s", source_file);
        (inFile = fopen(source_file, "r"))!=NULL;
        scanf("%s", source_file))
    {
        printf("无法打开源文件%s\n", source_file);
        printf("请重新输入文件名> ");
    }

    // 打开目标文件
    printf("请输入目标文件的名称> ");
    for(scanf("%s", dest_file);
        (outFile = fopen(dest_file, "w"))!=NULL;
        scanf("%s", dest_file))
    {
        printf("无法打开目标文件%s\n", source_file);
        printf("请重新输入文件名> ");
    }

    // 从源文件中读取后写入目标文件中
    for(ch=getc(inFile); ch!=EOF; ch=getc(inFile))
        putc(ch, outFile);
}
```




```
//关闭文件
fclose(inFile);
fclose(outFile);

//显示操作成功信息
printf("Copying %s to %s is succeed!\n", source_file, dest_file);

return 0;
}
```

请读者完成编码后编译并运行上述程序，运行结果如图 10.6 所示。

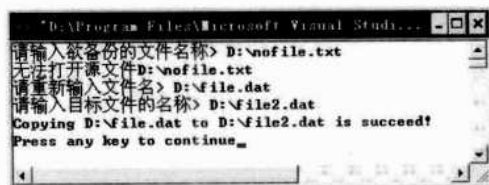


图10.6 程序运行结果

10.3.3 文件中字符串读/写

程序员可以实现文件内容的逐字符读取，但是读者可能会问这样操作会不会有些烦琐，那有没有一种更好的办法可以对文件进行读/写呢？事实上，C 语言不仅可以对文件中单个字符进行读/写，还可以直接对文件中的整个字符串进行读/写。本小节将就介绍这种文件操作方式。

1. fgets()函数

fgets()函数用来从指定的文件中读取指定长度的字符串，它定义在头文件 `stdio.h` 中，其调用形式如下：

```
fgets(str, n, pfile);
```

其中 `str` 是 `char` 型指针，指向一个将被赋值的数组名；`n` 是一个整型数，它表示准备读取的字符串的长度；`pfile` 为指向一个已经成功打开的文件或者标准输入设备文件 `stdin`。该函数返回一个指向该字符串的指针。

如果读到文件尾或出错，均返回一个空指针 `NULL`。所以在使用 `fgets()` 函数时，常用 `feof()` 函数来检测是否到达了文件尾（也可以用 `ferror()` 函数来测试是否出错，该函数在本章末尾介绍）。如果未到文件的结尾处，`fgets()` 函数将从打开的文件中读取 `n-1` 个字符，放入到 `str` 字符串数组中，并在 `str` 数组的最后增加一个空字符 “\0”。

通常把输出的字符赋予一个字符数组，构成赋值语句，例如：

```
char*str[3];
fgets(str, 3, pfile);
```

上面的程序把 `pfile` 指向的文件中 2 个字符读到 `str` 字符串中。`fgets()` 函数读取文件的过程可以用图 10.7 来说明。

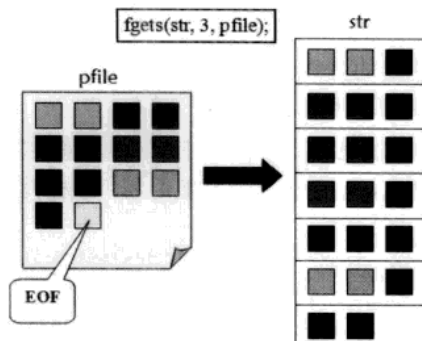


图10.7 使用fgetc()函数读取文件

举一个例子来说明 fgetc()函数的使用方法。该示例程序的作用是利用字符串读/写函数 fgetc()读取 file.txt 文件中的数据并在屏幕上显示出来。示例程序的实现代码清单如下：

```
#include <stdio.h>

int main()
{
    FILE *pfile;
    char str[128];
    if((pfile = fopen("D:\\file.txt", "r"))==NULL) //以只读方式打开文件
    {
        printf("无法打开文件!\n");
        return 0;
    }

    while(!feof(pfile)) //判断文件是否读取完毕
    {
        if(fgetc(str, 128, pfile)!=NULL) //判断是否成功读取字符串
        {
            printf("%s", str); //将字符串输出到屏幕上
        }
    }

    fclose(pfile);
    return 1;
}
```

在本示例程序中，文件被打开以后，程序在循环中利用 feof()函数来判断文件是否已经被读取完毕，并循环读取文件中内容，再在屏幕上打印出来。

在使用 fgetc()函数时还需要注意以下几点：

- ☑ fgetc()函数中第一个参数可以是定义的字符数组，也可以是动态分配的内存区，作为字符数组不用写数组名称后面的方括号以及其中的数组长度。
- ☑ 函数中第二个参数是“字符串实际长度+1”，因为字符串最后面还有一个/0位。
- ☑ fgetc()函数中第二个参数规定的字符串长度应该与字符串长度相等，否则运行程序时会有溢出的错误。当输入的字符串大于限定的字符串长度时，限定长度之后的字符串会被丢弃。
- ☑ fgetc()函数和 gets()函数不同，当读到“\n”不停止，仅把“\n”作为一个字符。



2. fputs()函数

fputs()函数的功能是向指定的文件写入一个字符串。它被定义在 `stdio.h` 中，使用时必须显式地包含此头文件。该函数的调用形式如下：

```
fputs(str, pfile);
```

其中 `str` 可以是字符串常量，也可以是字符数组名，或指针变量，`pfile` 是一个指向已经成功打开的文件。例如：

```
fputs("abcd", pfile);
```

本例的含义是把字符串“abcd”写入 `pfile` 所指的文件中。

下面举例说明该函数的用法。假设有一个文件 `file.txt`，现在向此文件当中追加一个字符串，并将文件内容显示在屏幕上。对于在文件中追加内容，首先应该以追加方式打开一个文件，然后以一定的格式向其中添入确定的内容，最后关闭文件。根据以上分析，该示例程序的具体实现代码如下：

```
#include <stdio.h>

int main()
{
    FILE *pfile;
    char ch, str[20];
    if((pfile=fopen("file.txt", "a+"))==NULL)    //以追加方式打开文件
    {
        printf("无法打开文件!");
        return 0;
    }

    printf("请输入一个字符串:\n");
    scanf("%s", str);                            //以格式化输入一个字符串
    fputs(str, pfile);                            //将字符串输入文件
    rewind(pfile);                                //将文件指针重新定位在文件开头
    ch = fgetc(pfile);                            //从文件中获取一个字符

    printf("\n");

    while(ch!=EOF)                                //判断是否到达文件末尾
    {
        putchar(ch);                             //将文件内容打印在屏幕上
        ch=fgetc(pfile);                         //继续获取下一个字符
    }

    printf("\n");
    fclose(pfile);
    return 1;
}
```

上述示例程序以追加读/写方式打开文件。然后在屏幕上由用户输入字符串，并用 `fputs()` 函数将该字符串写入文件。函数 `rewind()` 把文件位置指针移到文件开始处，这个函数在本章后续内容中还会详细介绍，这里就不再赘述了。程序最后将文件内容全部显示在屏幕上。请读者完成编码后自行编译运行程序。

3. fprintf()函数

fprintf()函数被定义在 stdio.h 文件中，使用时必须显式地包含此头文件。该函数的调用形式如下：

```
fprintf(pfile, 格式字符串, 输出项列表);
```

除第一个参数文件指针 pfile 外，其他参数与 printf()相同。实际上，fprintf()和 printf()在用法上基本相同，区别在于 printf()向控制台输出数据，而 fprintf()向文件中输出数据。在 fprintf()函数中，格式字符串用来指向控制串，它是必需的参数，指定字符串以及其中变量的格式定义。具体由以下 3 类字符组成：格式说明符、空白符和非空白符。格式字符串中包含的可能值如表 10.4 所示。

表10.4 fprintf格式字符串包含的常用符号

符 号	意 义
%%	返回百分号
%b	返回二进制数
%c	返回与 ASCII 值相对应的字符
%d	带有正负号的十进制数
%e	科学计数符号（如：1.2e+2）
%u	不带正负号的十进制数
%f	浮点数据（本地设置）
%F	浮点数据（非本地设置）
%o	十进制数
%s	字符串
%x	十六进制数（小写字母）
%X	十六进制数（大写字母）

输出项列表是被格式化到格式字符串中的字符，包括浮点型、整型、字符型等多种，每个字符之间以逗号（,）分开。例如：

```
fprintf( pfile, "%d %c", j, ch );
```

以上语句的作用是将整型变量 j 和字符型变量 ch 的值按照%d 和%c 的格式输出到 pfile 所指向的文件中。

4. fscanf()函数

fscanf()函数与 fprintf()函数一样，定义在 stdio.h 文件中，其调用形式如下：


```
fscanf(pfile, 格式字符串, 地址项列表);
```

与 scanf()函数一样，地址项表列是需要读入的所有变量的地址，而不是变量本身。其他各参数含义与 fprintf()函数相似，例如：

```
fscanf(pfile, "%d %c", &i, ch);
```

上述语句是从 pfile 所指向的磁盘文件中分别按%d 和%c 格式读入整型变量 i 和字符型变量 ch 的。



 注意：在 `fscanf()` 函数中，格式字符串中含有的空白部分与输入内容相似，这意味着 `tab` 键（制表符）和输入内容的一个单一空格相似。另外，`fprintf()` 和 `fscanf()` 函数用来格式化读/写的函数，与 `printf()`、`scanf()` 函数功能作用相似。而两者的区别主要在于 `fscanf()` 函数和 `fprintf()` 函数的读/写对象不是键盘和显示器，而是磁盘文件。

10.3.4 其他文件读/写函数

除前面介绍的那些函数外，C 语言还提供了其他一些文件读/写函数，它们同样都被定义在头文件 `stdio.h` 中，其中常见的有以下几个。

1. 字符读入函数

字符读入函数的形式为：

```
int getc(FILE * pfile);
```

该函数从 `pfile` 指针所指向的文件中读取一个字符，如果操作成功则返回该字符，如果文件结束或出错则返回 `EOF`。

2. 字符输出函数

字符输出函数的形式为：

```
int putc(int ch, FILE * pfile);
```

该函数把一个字符 `ch` 输出到 `pfile` 指针所指向的文件中，若成功则返回该字符的 ASCII 码，若失败则返回 `EOF`。

下面这段代码简单地标出了 `putc()` 函数的使用方法，注意 `stdout` 是一个标准设备文件，读者也可以把它换成普通文件，道理是相同的：

```
#include <stdio.h>

int main()
{
    char str[] = "Hello world!\n";
    int i = 0;

    while (str[i])
        putc(str[i++], stdout);

    return 0;
}
```

10.3.5 文件位置定位

前面介绍的文件读/写操作方法都具有一个共同的特点，那就是只能从文件起始处开始顺序地对文件内容进行读/写操作。这显然不够灵活，于是有人不禁会问有没有这样一种方法，它能够对文件内容直接定位并按照开发者的意愿直接修改文件中指定部分的内容。


C 语言提供了移动文件位置指针的函数来帮助开发人员实现这样的操作。实现文件位置指针的移动，也称为文件的定位。在 C 语言中实现文件位置指针移动的函数共有 3 个，分别是 `rewind()`、`fseek()` 和 `ftell()`。

1. rewind()函数

rewind()函数前面已多次使用过,下面给出它的函数声明原型,其中 pfile 是一个文件指针:

```
void rewind(FILE * pfile);
```

该函数的作用是将文件位置指针移到文件的开头,当移动成功时,函数返回 0,否则返回一个非 0 值。

 注意: 该函数只能将文件位置指针移动到文件开头,用户在使用该函数后,如果继续向文件中输入字符,将会覆盖原来文件开头的内容。

2. fseek()函数

fseek()函数用来移动文件位置指针到任意位置,下面给出它的函数声明原型:

```
int fseek(FILE * pfile, long offset, int base);
```

其中 pfile 是一个文件指针; offset 是位移量,表示移动的字节数,offset 是一个 long 型数据,以保证在文件长度大于 64KB 时不会出错; base 是起始点,表示从何处开始计算位移量,在 C 语言中,我们规定的起始点有 3 种:文件开始、当前位置和文件尾。Fessk()函数中起始点的取值如表 10.5 所示。

表11.5 fseek()函数中起始点取值表


起 始 点	表示符号	数字表示
文件开始	SEEK_SET	0
当前位置	SEEK_CUR	1
文件末尾	SEEK_END	2

例如,下面这行语句将把文件位置指针移到离文件开始处 100 个字节的地方:

```
fseek(pfile,100L,0);
```

而下面这行语句则把文件位置指针从文件尾向前移动 15 个字节:

```
fseek(pfile,-15L,2);
```

 注意: offset(位移量)参数要求是长整型数,当用常量表示 offset 时,一般加上后缀“L”。

另外, fseek()函数一般用于二进制文件,因为在文本文件中需要进行字符转换, fseek()函数计算的位置往往会出现错误。

图 10.8 示出了 rewind()函数和 fseek()函数的用法。

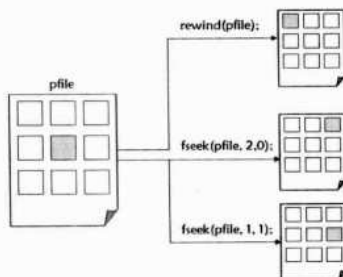


图10.8 文件定位函数的用法解析



3. ftell()函数

函数 `ftell()` 用来得到文件指针离文件开始处的偏移量，也就是说我们可以用它来获取文件的当前读/写位置，它的函数声明原型如下，其中 `pfile` 是文件指针：

```
long ftell(FILE * pfile);
```

该函数可以得到流文件当前的读/写位置，其返回值是当前读/写位置偏离文件头部的字节数，当该函数返回值是 -1 时表示函数执行出错。例如下面这段代码片段：

```
int position;
position = ftell( pfile );
if(position == -1L)
printf("文件读/写失败!\n");
```

该例获得了 `pfile` 文件中的当前读/写位置，并将其值传给变量 `position`。如果调用函数失败，则输出“文件读/写失败!”。

下面用一个示例来演示 `fseek()` 函数与 `ftell()` 函数的使用。这段代码的作用是求一个二进制文件的长度。分析可知，因为 `fseek()` 函数可以用来移动文件位置指针，`ftell()` 函数可以得到文件位置指针距离文件开头的偏移量，所以如果先把文件位置指针定位到文件末尾，然后求得指针在末尾时距离文件开头的偏移量就可以求出文件的总长度。下面给出具体实现的代码清单：

```
#include <stdio.h>

int main()
{
    FILE *pfile;
    long length;
    if((pfile=fopen("D:\\file.dat","rb")) == NULL)
        printf("Error on opening file!\n");
    else
    {
        fseek(pfile, 0L, 2);
        length=ftell(pfile);
        printf("The length of the file is %ld bytes.\n",length);
        fclose(pfile);
    }

    return 1;
}
```

请读者完成编码后，编译并运行程序观察输出结果。

10.3.6 数据块的读/写

在前文所述的 `fputc()` 和 `fgetc()` 函数可以用来读取文件中的一个字符，为了方便对整块数据进行读/写操作，ANSI C 标准还提出了两个函数——`fwrite()` 和 `fread()`，用来读/写一组数据（如一个数组元素或一个结构体变量的值等）。

它们的函数原型定义在头文件 `stdio.h` 中，在使用前必须包含头文件 `stdio.h`，它们的函数原型如下：


```
int fread(void * buffer, int size, int count, FILE *pfile);
int fwrite(void *buffer, int size, int count, FILE * pfile);
```

其中参数 `buffer` 是一个指针，它用来指向一个数据内存区域。在 `fread()` 函数中，`buffer` 是读入数据在内存中的起始地址；在 `fwrite()` 函数中，`buffer` 是输出数据在内存中的起始地址。参数 `size` 表示将要进行读/写的字节数。参数 `count` 表示将要读/写多少个 `size` 字节的数据项。`pfile` 是 `FILE` 类型指针。如果文件以二进制的形式打开，那么用 `fread()` 和 `fwrite()` 函数就可以读/写任何类型的信息，无论是数组还是结构体。

`fread()` 函数可以从指定的文件中读取 `count` 个数据项，每个数据项的长度为 `size` 个字节，读取的内容将存入由 `buffer` 指针指向的内存缓冲区中。图 10.9 示出了使用 `fread()` 函数读取文件时的行为。

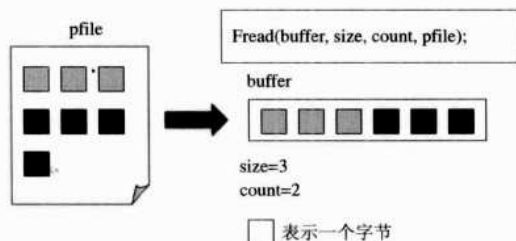


图10.9 使用fread()函数读取文件

在执行 `fread()` 函数时，文件指针会随着数据的读取而向后移动，最后移动结束的位置等于实际读出的字节数。该函数执行结束后，将返回实际读出的数据元素数，这个数据元素数不一定等于设置的 `count`，因为若文件中没有足够的数据元素项或读取过程中出错，都会导致返回的数据项数少于设置的 `count`。当返回数不等于 `count` 时，可以用 `feof()` 或 `ferror()` 函数进行检查。

`fwrite()` 函数从 `buffer` 指针指向的缓冲区中取出长度为 `count` 个 `size` 字节的数据项，写入到指针 `pfile` 所指向的文件中。执行该操作后，文件指针将向后移动，移动的字节数等于写入文件的字节数目。该函数操作完成后，也将返回成功写入的数据元素数。

关于函数 `fread()` 和 `fwrite()` 需要说明的是：

- ☐ 这两个函数读（写）的字节总数为 $\text{size} \times \text{count}$ ，其次，当这两个函数调用成功时，这两个函数各自返回实际读或写的数据元素项的个数，而不是字节总数。
- ☐ 当遇到文件结束或出错时，`fread()` 函数返回一个短整型值；当写出错时，`fwrite()` 函数也会返回一个短整型值。
- ☐ `fwrite()` 和 `fread()` 函数是按数据块的长度来处理输入/输出的，因此一般用于二进制文件的操作处理。

下面这段示例程序说明了函数 `fwrite()` 和 `fread()` 的用法。请读者自行编译并运行程序，限于篇幅这里就不再对其进行过多的解释了。

```
#include <stdio.h>
#include <string.h>
```




```
int main()
{
    char sentence[] = "world!";
    char buffer[20];

    FILE *pfile;
    if((pfile = fopen("D:\\file.dat", "a+"))==NULL)
    {
        printf("Cannot open the file!\n");
        return 0;
    }

    fwrite(sentence, strlen(sentence)+1, 1, pfile);
    int length = ftell(pfile);
    rewind(pfile);

    fread(buffer, length, 1, pfile);
    printf("%s\n", buffer);
    fclose(pfile);

    return 1;
}
```

10.4 处理二进制文件

1. 二进制文件的好处

读者应该明确文件在计算机中都是以二进制形式存储的，二进制对于机器是可见的，但对于人来说却是不可见的。反之亦然，人可见的形式是文本型的，但文本文件对于机器确实不可见的。

当使用文本文件存储数据时，程序就不得不花费相当多的时间将输入文件中的字符流转换为二进制流。同样，为了将数据存储在一个输出文本文件中，程序还必须再一次将内部数据格式（即二进制流）转换为字符流，这一过程又耗用了一部分宝贵的计算机时间，如图 10.10 所示。

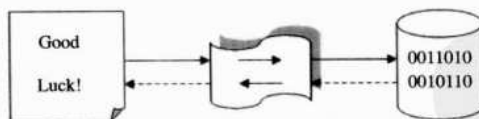


图10.10 文件形式的转换

文本文件仅仅是面向用户的，如果某些程序所产生的输出文件仅仅是作为其他程序的输入文件，而人们并不需要读取文件，那么显然将内部二进制数据格式转换为字符流无疑是在浪费计算机的时间，而且事实上这种情况非常多。

同样，应用一个反向转换在字符流中提取预期的文本数据也是浪费时间。为了避免这些不必要的转换，推荐使用二进制文件。使用二进制文件能够节约不必要的计算时间，这就是使用二进制文件的好处。

2. 二进制文件的缺点

二进制文件的不足就在于一台计算机上产生的二进制文件在另一种型号的计算机上有可能无法识别，这是由于二进制文件只能由特定的程序来读取。因此，人们也无法像在文字处理软件中检查和修改文本文件那样来处理二进制文件。二进制文件对于用户来说是不可见的。

3. 文本文件和二进制文件对比

下面对输入和输出不同类型数据的文本文件和二进制文件进行对比，见表 10.6。

表10.6 文本文件和二进制文件处理方式的对比

示例编号	处理文本文件	处理二进制文件	作 用
1	file1_in_t = fopen("file1.txt", "r"); file2_in_t = fopen("file2.txt", "r");	file1_in_b = fopen("file1.bin", "rb"); file2_in_b = fopen("file2.bin", "rb");	打开两个输入文件
2	file1_out_t = fopen("file3.txt", "w"); file2_out_t = fopen("file4.txt", "w");	file1_out_b = fopen("file3.bin", "wb"); file2_out_b = fopen("file4.bin", "wb");	打开两个输出文件
3	fscanf(file1_in_t, "%s%s%s%d%d", ticket.departure, ticket.destination, ticket.time, &ticket.num, &ticket.fee);	fread(&ticket, sizeof(train) 1, file1_in_b);	将列车车票结构从数据文件复制到内存中
4	fprintf(file1_out_t, "%s %s %s %d %d", ticket.departure, ticket.destination, ticket.time, ticket.time, ticket.fee);	fwrite(&ticket, sizeof(train), 1, file1_out_b);	将列车车票结构写入到输出文件中
5	for(i=0; i<SIZE, ++i) scanf(file2_in_t, "%d", &array[i]);	fread(array, sizeof(int), SIZE, file2_in_b);	用输入文件中的 int 型数值填充 array 数组
6	for(i=0; i<SIZE, ++i) scanf(file2_out_t, "%d\n", array[i]);	fwrite(array, sizeof(int), SIZE, file2_out_b);	将数组 array 的内容写入输出文件
7	fclose(file1_in_t); fclose(file2_in_t); fclose(file1_out_t); fclose(file2_out_t);	fclose(file1_in_b); fclose(file2_in_b); fclose(file1_out_b); fclose(file2_out_b);	关闭所有的输入和输出文件



4. 操作二进制文件的基本方法

该程序首先接收用户从键盘输入的数据，这些数据是 4 条关于火车票务的信息。然后程序将这些数据按照二进制的形式存入到磁盘文件中，磁盘文件名为 file1.bin。程序再从磁盘中将文件信息读出并打印到计算机屏幕上。以下是该示例程序的代码清单：

```
#include <stdio.h>

#define SIZE 4

typedef struct {
    char departure [20];    //列车始发站
    char destination [20]; //列车终点站
    char time[10];         //开车时间
    int num;               //列车车次
    int fee;               //乘车费用
} train;

train ticket[SIZE];
//将信息写入二进制文件
void writeInFile()
{
    FILE * file1_out_b;
    if((file1_out_b = fopen("file1.bin", "wb"))==NULL)
    {
        printf("Error on opening file!\n");
        return;
    }

    int i;
    for(i = 0; i < SIZE; i++)
        if(fwrite(&ticket[i], sizeof(train), 1, file1_out_b) != 1)
            printf("Error on writing file!");

    fclose(file1_out_b);
}

int main()
{
    int i;
    printf("请输入列车信息: \n 始发站 终点站 列车车次 开车时间 乘车费用\n");
    //接收键盘输入
    for(i = 0; i < SIZE; i++)
        scanf("%s%s%d%s%d",
            ticket[i].departure,
            ticket[i].destination,
            &ticket[i].num,
            ticket[i].time,
            &ticket[i].fee);

    writeInFile();
}
```

```
//将信息从二进制文件中读出并显示到屏幕上
printf("列车信息如下: \n 始发站 终点站 列车车次 开车时间 乘车费用\n");
FILE * file1_in_b;
file1_in_b = fopen("file1.bin", "rb");
for(i = 0; i < SIZE; i++)
{
    fread(&ticket[i], sizeof(train), 1, file1_in_b);
    printf("%s %s %d %s %d \n",
        ticket[i].departure,
        ticket[i].destination,
        ticket[i].num,
        ticket[i].time,
        ticket[i].fee);
}

fclose(file1_in_b);

return 1;
}
```

请读者完成编码后编译并运行上述程序, 图 10.11 是该程序的运行结果。



图10.11 程序运行结果

10.5 文件缓冲区处理

通常在文件读/写操作过程中, 只有当执行 `fclose()` 函数关闭文件时, 缓冲区中的内容才会被写入磁盘文件。可是读者可能会问有没有办法手动清除缓冲区数据, 或者使缓冲区的数据及时被写入到磁盘文件中? 如何使文件按缓冲区按照程序设计者的意愿来工作将是本节的重点讨论内容。

10.5.1 文件缓冲区的清除

C 语言中对缓冲区的清除主要使用 `fflush()` 和 `fflushall()` 两个函数, 它们都被定义在头文件 `stdio.h` 中, 以下分别对这两个函数进行介绍。

1. `fflush()` 函数

`fflush()` 函数的声明原型如下:



```
int fflush(FILE * pfile);
```

该函数将清除由 `pfile` 指向的文件缓冲区里的内容。该函数可以帮助程序员在写完一些数据后，立即手动地清除缓冲区，以免误操作时，破坏原有数据。当文件以写方式打开时，缓冲区的内容将被写入文件。图 10.12 演示了函数 `fflush()` 的作用。

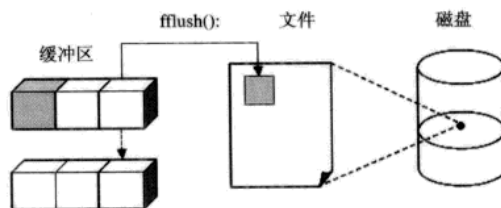



图10.12 函数fflush()的作用

请读者来看下面这段示例代码：

```
#include <stdio.h>

int main()
{
    FILE *pfile;
    pfile = fopen("D:\\data.txt", "w+");
    fputc('c', pfile);
    //!!!!此处将把文件缓冲区内容写入磁盘文件中
    //fflush(pfile);
    getchar();
    fclose(pfile);
    return 1;
}
```

下面我们一同来看看上述程序的执行结果如何。首先请读者编译并运行上述代码，当程序由于语句“`getchar();`”而暂时停滞的时候，读者可以在磁盘上找到文件“`D:\\data.txt`”，此时打开该文件不难发现，文件中没有任何字符。按任意键，程序继续运行直到结束，这时再次打开文件“`D:\\data.txt`”，即可发现字符 `c` 被写入到文件中了。

 **提示：**为了实验语句“`fflush(pfile);`”的作用，请读者将上述代码中该行语句的注释符号去掉，再次编译并运行程序，当程序由于语句“`getchar();`”而暂时停滞的时候，读者可以在磁盘上找到文件“`D:\\data.txt`”，结果与前面的实验结果完全不同，我们发现字符 `c` 在文件还未被关闭之时已被写入其中了。

2. fflushall()函数

`fflushall()` 函数的声明原型如下：

```
int fflushall(void);
```

`fflushall()` 函数与 `fflush()` 函数功能相同，都是用来清除缓冲区中的内容，但是 `fflushall()` 函数将清除所有打开文件所对应的文件缓冲区。

10.5.2 文件缓冲区的设置

C 语言中对文件缓冲区进行设置主要使用 `setbuf()` 和 `setvbuf()` 两个函数, 它们都被定义在头文件 `stdio.h` 中。

1. `setbuf()` 函数

`setbuf()` 函数的声明原型如下:

```
void setbuf(FILE *pfile, char *buffer);
```

该函数的作用简单来说就是把缓冲区与流相关联。我们已经知道通常 C 程序控制输出的方式有两种:

- ☑ 收到数据立即处理。
- ☑ 先暂存到缓冲区后, 然后再大块写入。

由于前者将有可能引起过渡频繁的操作, 而给系统造成了较高的负担。因此, C 语言通常也允许程序员在进行实际的写操作之前控制产生的输出数据量。`setbuf()` 函数将使所有写入到 `pfile` 的输出都应该使用 `buffer` 作为输出缓冲区 (`buffer` 是一个大小适当的字符数组), 直到该缓冲区被填满或者程序员直接调用 `fflush` 时, 缓冲区中的内容才实际被写入到 `pfile` 所指向的文件实体中。

请读者来看下面这段示例程序。该程序首先打开名为 “D:\\data1” 和 “D:\\data2” 的两个文件, 然后又使用 `setbuf()` 函数为第一个文件赋予一个用户定义的缓冲区, 并使第二个文件的缓冲区为空。

```
#include <stdio.h>

int main()
{
    char buf[BUFSIZ];
    FILE *file1 = fopen("D:\\data1", "a");
    FILE *file2 = fopen("D:\\data2", "w");

    if((file1 != NULL)&&(file2 != NULL))
    {
        setbuf(file1, buf);
        printf("file1 set to user-defined buffer at: %Fp\n", buf);

        setbuf(file2, NULL);
        printf("file2 buffering disabled\n");
    }

    fclose(file1);
    fclose(file2);
    return 1;
}
```

上述程序中的 `BUFSIZ` 用来指定缓冲区的大小, 它是包含在头文件 `stdio.h` 中的一个定义。请读者完成编码后编译并运行上述程序, 结果如图 10.13 所示。

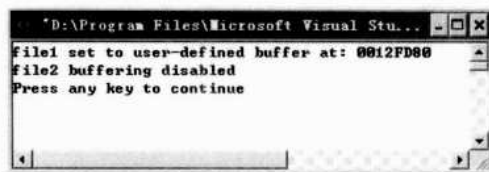


图10.13 程序运行结果

2. setvbuf()函数

函数 `setvbuf()` 也用于设置文件缓冲区，功能和 `stebuf()` 有些相近，但是该函数所含参数较多，功能更为复杂。其声明原型如下：

```
int setvbuf(FILE * pfile, char *buffer, int type, unsigned size);
```

其中 `pfile` 为指向文件的指针，`buffer` 是指向的缓冲区的指针，`size` 用于指定缓冲区的大小，`type` 是类型，其值如表 10.7 所示。

表10.7 setvbuf()函数中type值的含义

type 值	意 义
<code>_IOFBF</code>	文件是完全缓冲区，当缓冲区是空时，下一个输入操作将企图填满整个缓冲区。在输出时，在把任何数据写到文件之前，将完全填充缓冲区
<code>_IOLBF</code>	文件是行缓冲区。当缓冲区为空时，下一个输入操作将仍然企图填满整个缓冲区。然而在输出时，每当新行符写到文件，缓冲区就已被清洗掉了
<code>_IONBF</code>	文件是无缓冲的， <code>buf</code> 和 <code>size</code> 参数是被忽略的。每个输入操作将直接从文件中读，每个输出操作将立即把数据写到文件中

下面这段代码示出了 `setvbuf()` 函数的基本用法，请读者完成编码后编译并运行程序。

```
#include <stdio.h>

int main()
{
    char bufr[10];

    FILE *data = fopen("D:\\file.txt", "w");

    if (setvbuf(data, bufr, _IOFBF, 10) != 0)
        printf("Failed to set up buffer for input file.\n");
    else
        printf("Buffer set up for input file.\n");

    int i = 0;
    for(; i < 11; i++)
        fputc('a' + i, data);
    getchar();

    fclose(data);
    return 0;
}
```


下面来分析一下输出结果。首先,上述程序使用 `setvbuf()` 函数为文件指定了一个大小为 10 字节的缓冲区。然后程序又向文件缓冲区中输入了 11 个字符。由于缓冲区仅有 10 个字节的大小,即仅能容纳 10 个字符,所以当缓冲区满时,程序就会将缓冲区中的内容写入磁盘文件中,并清除缓冲区,所以在语句“`getchar();`”使程序停滞的时候,读者可以看到文件“D:\file.txt”的内容如图 10.14 所示。

当用户按任意键使程序继续运行后,程序调用 `close()` 函数将文件关闭,程序运行结束。此时,我们再次打开文件“D:\file.txt”,该文件的内容即如图 10.15 所示,可见此时第 11 个字符才被输入。

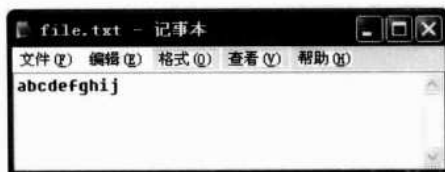


图10.14 程序运行的中间结果

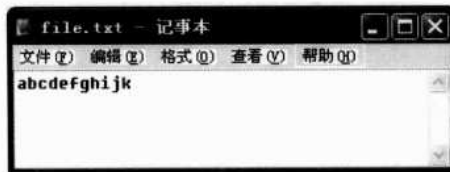


图10.15 程序运行的最终结果

10.6 文件操作的检测

在文件打开、关闭以及读/写操作过程中,有时可能会发生一些错误,C语言标准中提供了一些用于检测输入/输出错误的函数。C语言对文件操作检测的常用函数最基本的有 3 个,它们均定义在头文件 `stdio.h` 中。

1. `feof()` 函数

在对文件进行读/写操作的过程中,经常需要判断文件是否已经读/写结束,`feof()` 函数在前面已经多次提到,它就是用来测试文件位置指针是否达到文件尾的。`feof()` 函数原型定义在头文件 `stdio.h` 中,其具体调用形式为:

```
feof(pfile);
```

其中 `pfile` 必须是一个有效的文件指针,而且该文件必须已经成功打开。本函数可以测试文件位置指针是否已经指向文件尾,若已到达文件尾或发生错误则返回 `True`,其他情形返回 `False` 值。

2. `ferror()` 函数

函数 `ferror()` 用来检测指定流上读/写的错误,它的一般调用形式为:

```
ferror(pfile);
```

与上述函数相同,其中的 `pfile` 指向一个已经打开的文件。如果流错误标志被置位,它将保持不变,直到调用 `clearerr()` 或 `rewind()` 函数,或者关闭为止,如果检测到指定流上的错误,将返回非 0 值。特别地,在执行 `fopen()` 函数时,`ferror()` 函数的初始值会被自动设为 0。

3. `clearerr()` 函数

`clearerr()` 是复位错误标志函数,它的功能是使文件错误标志和文件结束标志置为 0。它的一般调用形式为:



```
clearerr(pfile);
```

如果错误标志被置位，流的操作将不断返回错误状态，直到调用 `clearerr()` 或 `rewind()` 函数。该函数没有返回值。与 `ferror()` 函数相反，`clearerr()` 函数主要用来清除 `pfile` 指向文件流的错误标志。因此在 `pfile` 指向的流出错之后可以由 `clearerr()` 来清除错误标志。

4. `ferror()` 和 `clearerr()` 函数用法

下面这段示例程序说明了 `ferror()` 函数和 `clearerr()` 函数的用法：

```
#include <stdio.h>

void main()
{
    FILE * pfile;
    char ch;
    pfile = fopen("file.dat", "w");
    ch = fgetc(pfile);
    printf("%c\n", ch);

    if(ferror(pfile))
    {
        printf("从 file.dat 文件中读取数据出错!\n");
        clearerr(pfile);
        if(!ferror(pfile))
        {
            printf("提示: 文件错误标志已经复位!\n");
        }
    }
    fclose(pfile);
}
```

完成编码后，编译并运行程序，结果如图 10.16 所示。这段示例程序首先从 `file.dat` 文件中读取数据，但是由于 `file.dat` 文件并不存在，所以当使用 `ferror()` 函数判断读取数据是否失败时，它将返回非零值，所以程序就会输出文件读取失败的信息。然后程序再调用 `clearerr()` 函数将文件错误标志复位，所以再次使用 `ferror()` 函数时返回值就是零值了。

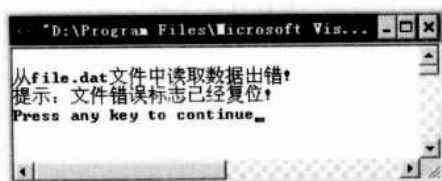


图10.16 程序运行结果



第 11 章

动态数据结构



在程序设计语言中，每一个数据都属于某种数据类型。类型显式或隐式地规定了数据的取值范围、存储方式以及允许进行的运算。这些已经事先定义好的数据类型就是所谓的原子型。然而，当今世界纷繁复杂，人们每天面对着林林总总的事物，信息从各个方面以多种多样的形态呈现在人们面前，仅仅使用原子型数据很难有效地抽象现实问题。

因为人们所面对的数据量有时是非常巨大的，所以如果没有一个有效的方法来储存、组织和表示信息，那么信息将变得没有任何价值。即使面对一组相对较小的信息时，一个结构化的数据表示方法也是必不可少的。假想公共汽车站等车的人不按顺序争先恐后地抢着上车，那么场面将是多么混乱！相反，如果人们排着队井然有序地逐个上车情况就好多了。

将原子型数据按照一定规则重组，就可以形成结构型数据，将原子型数据组织成结构型数据的过程中所使用的模型，就是数据结构。或者说，数据结构就是一个信息的结构化表示，是为了将原子型数据组织成结构化数据而套用的某种组织模型。

前面曾经讲过的数组就可以被认为是一种最简单、最朴素的数据结构。但是使用数组的一个问题在于，它是定长的静态的数据结构。而现实中的很多问题都是随着客观情况的变化而动态改变的。因此使用一个静态的数组来处理问题总是会遇到一些问题。为了解决这种矛盾，人们引入了动态数据结构的概念。在程序执行的过程中，能够动态地扩展或收缩的数据结构，就称为动态数据结构。本章就研究使用 C 语言来组织动态数据结构的方法。



11.1 动态内存管理

C 语言中动态内存管理为实现动态数据结构提供了语法支持。在 C 语言中，动态内存管理是通过库函数来完成的，本节就介绍 C 语言中的动态内存管理方法。

11.1.1 为什么使用动态内存分配

1. 数组数据存储的弊端

在使用数组时，系统会先在内存中开辟一块连续的区域，然后把数据依次存储进去。当访问其中的一个或几个数据时，可以利用这些数据的索引进行操作。这样的数据结构虽然实现简单、操作方便，但也致使程序的可伸缩性大大降低。

例如现在需要将一幅数字图像的像素矩阵存入数组，那么在不知道该图像的具体信息之前通常这个数组需要被定义得非常大以便能够容下任意大小的图像数据。然而当图像较小时，显然会造成内存的浪费，并且一旦程序载入一幅足以超出数组容量的大图片时，那么就可能导致程序崩溃。

2. 程序员使用数组时遇到的麻烦

在很多的情况下，程序员无法预先确定要使用多大的数组，因为数组中可能存放的是一个班级里的所有学生，也可能是用来存储这些学生的名字。由于并不知道该班级的学生的具体人数，所以程序员就不得不把数组定义得足够大。这样，程序在运行时就申请了固定大小的本以为足够大的内存空间。如果实际人数没有预想的那么大，结果是空间被浪费了。另外有些时候尽管本以为足够大的数组也有可能因为客观情况的突变而不再适用。

例如一个用于存储人名的字符数组，因为汉人的名字一般是 3 个字，最多不超过 3 个字（如果有复姓的话），于是我们就想当然地定义了一个能够存储 4 个字的数组。结果有一天来了一个外国人，他的名字可能很长。于是程序员又不得不重新去修改程序，扩大数组的存储范围。

3. 静态与动态内存分配

数组分配固定大小的内存分配方法称之为静态内存分配。但是这种内存分配的方法存在比较严重的缺陷。正如上面所列举的那些例子一样，在大多数情况下定长意味着会浪费宝贵的内存空间，而在某些特殊情况下，当定义的数组不够大时，可能引起下标越界错误，甚至导致严重后果。

为了解决这样的问题，就需要使用动态内存分配。所谓动态内存分配就是指在程序执行的过程中动态地分配或者回收存储空间的分配内存的方法。由于动态分配不需要预先分配存储空间，而且分配的空间还可以根据程序的需要扩大或缩小，因此它的确可以解决静态内存分配所带来的种种弊端。

11.1.2 如何实现动态内存管理

C 语言编译系统的库函数提供了一系列用于实现动态内存管理的函数，ANSI 标准建议的 4 个有关动态内存管理的函数有：

☑ malloc()。

☑ calloc()。

☑ realloc()。

☑ free()。

1. malloc()函数

malloc()函数原型为：

```
void * malloc(unsigned int size);
```

malloc()函数的作用是在内存的动态存储区中分配一个长度为 size 的连续空间。其参数是一个无符号整形数，返回值是一个指向所分配的连续存储域的起始地址的指针（类型为 void）。必须注意的是，当函数未能成功分配存储空间（如内存不足）就会返回一个 NULL 指针。关于分配失败的原因，可能有多种，比如说空间不足就是其中一种。所以在调用该函数时应该检测返回值是否为 NULL 并执行相应的操作。



注意：虽然这里的存储块是通过动态分配得到的，但是它的大小也是确定的，也就是通过参数 size 来限定的。因此也不允许越界使用。越界使用动态分配的存储块，尤其是越界赋值，可能引起非常严重的后果，而使程序崩溃。

下面给出一个应用 malloc()函数进行动态分配的示例程序：

```
#include <stdio.h>
#include <malloc.h>

void main()
{
    int count,*array;
    array = (int *)malloc(10*sizeof(int));
    if(array == NULL)
    {
        printf("不能成功分配存储空间!");
        return;
    }

    //给数组赋值
    for (count=0; count<10; count++)
        array[count]=count;
    //显示赋值结果
    for(count=0; count<10; count++)
        printf("%d",array[count]);
    printf("\n");
    free(array);
}
```

上述示例程序动态分配了 10 个整型存储区域，然后将赋值结果输出到屏幕上。其中，变量 array 是一个整型指针，也可以理解为指向一个整型数组首地址的指针。



2. calloc()函数

calloc()函数的原型为:

```
void * calloc(unsigned int n, unsigned int size);
```

calloc()函数的作用是在内存的动态存储区中分配 n 个长度为 $size$ 的连续空间。该函数返回一个指向分配域起始地址的指针;如果分配不成功,则返回 NULL。用 calloc()函数可以为二维数组开辟动态存储空间, n 为数组元素个数,每个元素长度为 $size$ 。例如下面这段示例程序简单地说明这一用途:

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    int *array = (int*)calloc(10, sizeof(int));
    for(int i = 0; i < 10; i++)
        array[i]=i;

    printf("%d\n", array[5]);
    printf("%d\n", array[9]);
    free(array);
}
```

请读者完成编码后自行运行上述程序并观察输出结果。

3. realloc()函数

realloc()函数的作用是修改已经分配的动态内存区域的大小,它的原型为:

```
void * realloc(void * p, unsigned int size);
```

该函数将 p 所指内存区域大小改为 $size$ 。 $size$ 可以比原来分配的空间大或者小。函数的返回值是指向该内存区域的指针。realloc()函数的用法比较简单,这里不再举例。

4. free()函数

由于内存区域总是有限的,因此不能不限制地分配下去。而且一个程序要尽量节省资源,所以当所分配的内存区域不用时,就要释放它,以便其他的变量或者程序使用,这时就要用到 free()函数了。无论是由 malloc()函数来申请的内存空间,还是由 calloc()函数来申请的内存空间都需要使用 free()函数来进行释放。

free()函数的原型为:

```
void * free(void * p);
```

其作用是释放指针 p 所指向的内存区域,从而使这部分内存区域能够被其他变量或者程序所使用。其参数 p 是最近一次调用 malloc()函数或 calloc()函数(另一个动态分配存储区域的函数)时返回的指针。另外,该函数无返回值。

在使用 free()函数时,需要注意,释放的内容是指针所指向的具体值,而不是用来申请动态内存的指针本身。例如下面这段示例代码:

```
int *array1,*array2;
array1= (int *) malloc (10*sizeof(int));
```



```
array2=array1;
...
free(array2) /*或者 free(array1)*/
```

注意上述代码中, malloc()函数的返回值被赋给了 array1, array1 的值又被赋给了 array2, 所以此时 array1 和 array2 同指向一块区域, 因此都可作为 free()函数的参数。而且 free(array2) 和 free(array1)是等价的, 因此只能使用它们中的一种, 如果两个语句都被使用的话就会产生重复释放的错误, 这点特别需要注意。

11.1.3 关于动态内存分配的说明

动态内存分配非常灵活而强大, 但也因此并不容易被掌握。使用不当就会造成内存泄漏等问题。这里对 C 语言中的动态内存分配做一些说明。

1. 使用 malloc()时需注意的问题

- ❑ 由于分配操作可能不成功, 因此申请了内存空间后, 必须检查是否分配成功。其次, 当不再需要使用已经被分配的内存时, 需要使用 free 释放; 释放后应该把指向这块内存的指针指向 NULL, 防止程序后面不小心使用了它。
- ❑ malloc()和 free()这两个函数应该是配对的。如果申请后不释放就会产生内存泄露; 释放只能一次, 如果释放两次及两次以上就会产生内存错误(释放空指针例外, 释放空指针其实等于什么都没做, 所以释放空指针释放多少次都没有问题, 但这样做是没有意义的)。
- ❑ 虽然 malloc()函数的类型是 void *, 任何类型的指针都可以转换成 void *, 但是处于安全考虑, 最好还是在前面进行强制类型转换。

2. 注意使用 sizeof 运算符

上一小节的例子中我们用到了 sizeof 运算符, 这里对此进行简要说明。有时候, 程序员可能需要知道某种类型的数据包含了几个字节。然而更多的情况下, 由于 C 语言中内置数据类型的大小是随运行环境的不同而不同, 在所有情况下要知道一个变量的大小可能是很困难的。

为了解决这个问题, C 语言定义了编译时运算符 sizeof。sizeof 运算符的书写形式与函数类似, 但它不是函数, 而是运算符, 而且是单目运算符, 优先级为 2 级, 可以与其他运算符一起组成表达式, 如 x*sizeof(int)。所谓编译时运算符, 是指程序在编译时就已经知道了变量或数据类型在内存中将占据多少字节。sizeof 的基本用法如下:

```
sizeof(操作数)
```

其中, 操作数可以是一个表达式或者是一个类型名, 如果操作数是一个数据类型的类型名, 那么 sizeof 将返回指定数据类型的大小, 如果操作数是一个表达式, 那么 sizeof 将返回指定值的大小, 注意这个“值”指的是表达式最终运算结果在内存中所占据的字节数。注意, 如果想知道某种数据类型的大小(例如 int 或 char), 那么必须将类型的名字包含在圆括号中。然而, 如果想知道某个值在内存中的大小, 圆括号就不是必需的, 但使用圆括号也是可以的。

3. 关于 free()函数的理解

这个问题其实前面已经提到过了。free()函数释放的是指针指向的内存, 而不是指针。这点非常重要! 指针是一个变量, 只有程序结束时才被销毁。释放了内存空间后, 原来指向




这块空间的指针仍然存在。只不过现在指针指向的内容是未定义的。因此，释放内存后最好把指针指向 NULL，防止指针在后面不小心被误用。

4. 理解堆空间


动态内存分配是在堆空间上实现的。也就是说使用 `malloc()` 函数后返回的指针指向的是堆空间里面的一块内存。堆是系统中的一块共有空间，分全局堆和局部堆。全局堆就是所有没有被分配的闲置空间，局部堆就是用户已分配的空间。

堆在操作系统对进程（程序的运行实体）初始化的时候分配，运行过程中也可以动态地向系统要额外的堆空间。但是被申请的堆在用完后，必须被释放，否则就会产生内存泄漏。

 **注意：**数据结构中也有一个“堆”的概念，但它与我们这里说的“堆”是两回事。

5. 理解栈空间

和堆相对应的另外一块空间是栈。栈是线程独有的，保存其运行状态和局部变量的空间。栈在线程开始的时候被初始化，每个线程的栈都彼此独立。每个函数都有自己的栈，栈被用来在函数之间传递参数以及保存现场等，因此有时也称活动记录栈。操作系统在切换线程的时候会自动切换栈。栈空间不需要在高级语言里面显式地分配和释放。

 **注意：**这里的栈是指内存中的一块区域，跟本章后面讲的数据结构中的“栈”不是同一概念）。

如果在函数上面定义了一个指针变量，然后在这个函数体内为其动态分配存储空间并让指针指向该区域。那么，这个指针的地址是在栈上，所以当函数返回后，该指针就不再存在了。但是它所指向的内容却是在堆上面的，因此，即使函数已经返回，该堆空间仍被占用，且无法释放。如下面这段示例代码：

```
//...
void Function(void)
{
    char *p = (char *)malloc(100 * sizeof(char));
}
```

在上述代码中，在函数返回后，函数所在的栈被销毁，因此指针 `p` 也跟着被销毁。但是因为申请的内存存在堆上，而函数所在的栈被销毁并不表示堆也跟着被销毁。所以，这就会导致内存泄漏。请读者注意，必须有 `free()` 函数和 `malloc()` 函数相对应，以确保堆空间能够被有效释放。

11.2 链表概述

链表是一种常见的线性数据结构。它被定义为一个结点序列，其中除了最后一个结点外的每个结点都包含有下一个结点的地址。整个序列就好像一串珠链一样，呈“线性”排列。实际链表有很多种。例如：单向链表、双向链表和循环链表等。本节就以其中最简单的单向链表为例来向读者介绍有关链表的一些基本知识，注意如果没有特别说明，文中出现的链表就是指单向链表。

11.2.1 单向链表与数组

在单向链表中元素的存放呈“线性”结构。单向链表与数组在很多方面都有相似之处，例如这两种数据结构都支持对元素的删除、遍历以及其他的一些基本操作。但数组与单向链表在完成一些特殊操作的效率上却存在很大差别。

例如，在单向链表中可以很方便地移动第一个元素，但在数组中却不可能做到这样。另一方面，数组可以对元素进行随机访问，但单向链表却不可以。这些差别是由数组与单向链表在内存中存储方式的不同造成的。有关单向链表在内存中的存储方式将在稍后介绍。

11.2.2 单向链表——老鹰捉小鸡

现实中一个能够比拟成单向链表的例子就是“老鹰捉小鸡”。这个游戏相信大家一定都玩过。如图 11.1 所示，在这个游戏中，除了有一个孩子扮演老鹰，一个孩子扮演老母鸡以外，其他孩子都是小鸡。来看看母鸡和小鸡之间的连接关系：

- ☑ 母鸡冲在队伍的最前。
- ☑ 然后有一只小鸡仅仅拉住母鸡。
- ☑ 再后面的小鸡将依次拉住前面的小鸡。

这样就形成了一个连续的线性的结构。

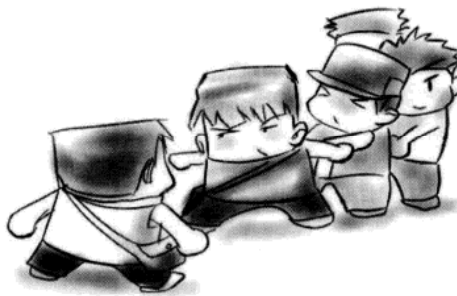


图11.1 老鹰捉小鸡

链表亦是如此，每个单向链表都有：

- ☑ 一个表头指针 (head)。
- ☑ 一个表尾指针 (tail)。

表头指针指向表头结点。表头结点是单向链表中第一个结点的前驱，它存放一个地址。该地址指向一个元素，链表中的每个元素被称为“结点”。每个结点都包括两个部分，即实际数据和下一个结点的地址。表尾指针指向单向链表的最后一个结点，这个结点不再指向其他结点，它称为表尾，存储的指针值为 NULL。表头指针标识了单向链表的开始，表尾指针标识了单向链表的结束。

由此可见，要找到某一个单向链表，只需找到它的表头指针即可。要找到某一个结点，则必须先找到它的前驱结点，然后再由前驱结点存储的指针找到该结点。如果某一个结点存储的指针值为 NULL，则这个结点为表尾，单向链表到此结束。单向链表的结构如图 11.2 所示。

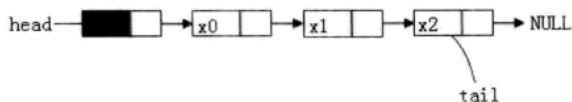


图11.2 单向链表的结构

单向链表在内存中的存放并不像图 11.2 所示那样是“连续”的。实际上单向链表的各个结点分散存储在内存中的不同区域，然后通过指针把这些结点“连接”起来。当对一个单向链表进行遍历时，先找到它的表头结点，然后根据表头结点中存储的指针找到下一个结点，如此循环下去直到遍历完成。

11.2.3 链表存储方式优缺点

链表在内存中非连续的存储方式有优点也有不足。

1. 优点

这种存储方式有利于结点的删除与插入，因为只要简单地更改几个指针就可以了。比如，用这种方法可以很容易地删除图 11.3 所示的链表中的第二个结点。具体过程就是令原来指向第二个结点的指针指向第三个结点。

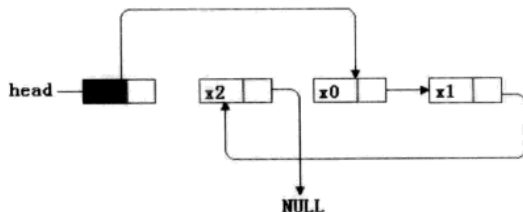


图11.3 链表的实际存储

2. 缺点

这种存储方式不利于对链表中的结点进行随机访问。为了访问链表中某一个指定的结点，必须从链表头开始遍历整个链表直到找到要访问的结点为止。链表的不连续的存储方式使程序不能仅依靠计算得到结点的地址，这与数组截然不同。数组利用内存中连续的区域存储数据，其中任何数据的地址都可以根据数组的首地址计算获得。

11.2.4 不同单向链表间的合并

表头指针与表尾指针除了标识单向链表的开始与结束，还可以用于不同单向链表间的合并。当要合并两个单向链表时，只需要根据表尾指针找到一个单向链表的表尾，再使这个表尾存储的值为 NULL 的指针指向另一个单向链表的表头即可。

由于表头中并不包含实际数据，因此第二个链表的表头可以不用，而使第一个链表表尾结点中的指针指向第二个链表的除表头结点以外的第一个指针即可，整个过程如图 11.4 所示。

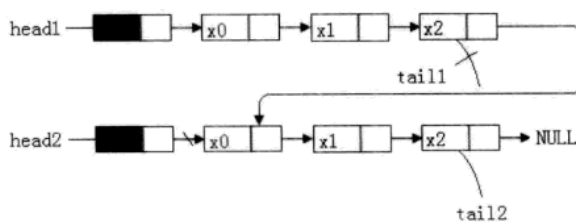


图11.4 单向链表间的合并

11.3 链表的操作及实现

C 语言中使用结构体变量来表示链表中的结点最为合适。下面给出了结构体表示的链表，结构体 `node` 中不但给出了该结点所存储的数据 `data`，还给出了指向下一个结点的指针 `next`。本小节就以此为基础来介绍有关链表的一些操作及其实现方法。

```
struct node
{
    int data;
    struct node * next;
};
```

11.3.1 链表的建立

1. 动态地建立链表

如何动态地建立一个链表呢？建立链表的基本方法分两个步骤。

第1步 动态地分配结点。

第2步 将已经生成的结点彼此连接起来。

例如，如果我们想使用链表来存储 1、10 和 100 这 3 个数据，那么可以使用如下的语句来分配并初始化 3 个结点的数据成员：

```
node *p1, *p2, *p3;
p1 = (node *)malloc(sizeof(node));
p1->data = 1;
p2 = (node *)malloc(sizeof(node));
p2->data = 10;
p3 = (node *)malloc(sizeof(node));
p3->data = 100;
```

经过上述的初始化过程，3 个结点已被成功分配，结果如图 11.5 所示。

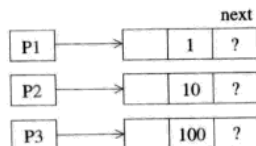


图11.5 初始化结点

接下来再对 3 个孤立的结点进行串接，如图 11.6 所示，将指针 `p1` 所指向结点中的 `next`



指针指向第二个结点，将指针 p2 所指向结点中的 next 指针指向第三个结点。然后将指针 p1 作为链表的入口，也就是头指针，并将第三个结点的 next 指针赋值为 NULL，这样它就变成了链表的尾结点，一个链表就生成了。

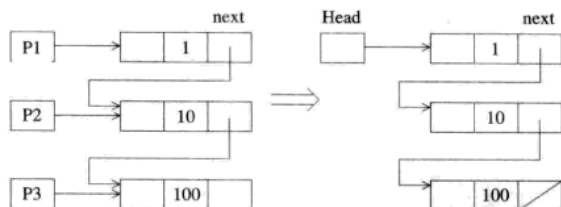


图11.6 链表结点的串接

结点连接的代码如下：

```
p1->next = p2;
p2->next = p3;
p3->next = NULL;
```

通过以上几个简单的步骤，一个链表就建立完成了。

2. 重复利用指针来建立链表

实际中链表的长度无法固定，初始化时为每个结点都建立一个指针变量显然是非常不经济的。更多的时候，往往是少数几个指针交替使用来建立链表。一般情况下，需要用到的指针变量应为 3 个，即 head、p1 和 p2，其中 head 指针指向链表头结点，p1 和 p2 指针则是用来重复使用的临时指针变量。下面我们来看看如何重复利用这 3 个指针建立链表。

第 1 步 如果链表为空链表，那么就可以直接将 head 指针赋为 NULL，而无须进行其他操作。

第 2 步 如果链表不为空，那么首先使用 malloc() 函数新建一个结点，使 head、p1 和 p2 都指向它。它就是这个链表的头结点。

第 3 步 对结点内的数据进行赋值，从而完成该结点的初始化工作。

第 4 步 开辟另外一个结点，并使 p1 指向这个新开辟的结点。

第 5 步 如果链表还未满足结束条件，那么继续将这个新的结点链入链表，也就是将 p1 的值赋给 p2->next（注意这个时候 p2 仍指向第一个结点），所以在执行了语句 p2->next=p1 之后，新结点就被成功链入链表了。

第 6 步 让指针 p2 向后移动一个位置，即执行语句 p2=p1，也就是使 p2 指向最新建立的结点。上述过程如图 11.7 所示。

第 7 步 再次开辟一个新结点并初始化它。

第 8 步 再次将指针 p1 指向这个新结点。如果链表仍然未满足结束条件，那么继续将这个新的结点链入链表，也就是将 p1 的值赋给 p2->next（注意这个时候 p2 仍指向前一个结点），所以在执行了语句 p2->next=p1 之后，新结点同样被成功链入链表。

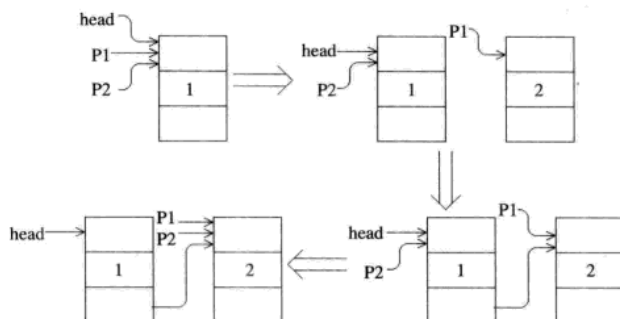


图11.7 链表的建立1

第9步 再像前面一样将指针 $p2$ 向后移动一个位置，即执行语句 $p2=p1$ ，也就是使 $p2$ 再次指向最新建立的结点。上述过程如图 11.8 所示。

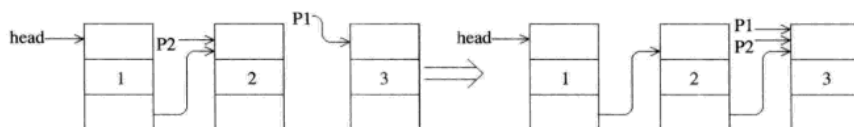


图11.8 链表的建立2

注意：为链表不断链入新结点的过程其实就是一个不断将指针 $p1$ 和 $p2$ 向后移动的过程。

第10步 如果仍有新的数据被接收，那么就再次为该数据开辟一个新结点，并让指针 $p1$ 指向这个新结点，并依照前面的操作方式继续为链表添加新数据。

第11步 如果链表已经满足结束条件（例如接收到了一个结束标志）或者达到事前预定的长度，那么循环操作就将被终止。新的结点也将不再被链入链表中。这个时候，即将 $NULL$ 赋给 $p2 \rightarrow next$ ，表示该结点为整个链表的尾结点。

链表的建立工作到此全部结束。上述过程如图 11.9 所示。现在请读者根据上面的描述，先自己尝试编写一个动态建立链表的函数。在下一小节中，我们将把这个函数同链表的遍历函数一同使用。

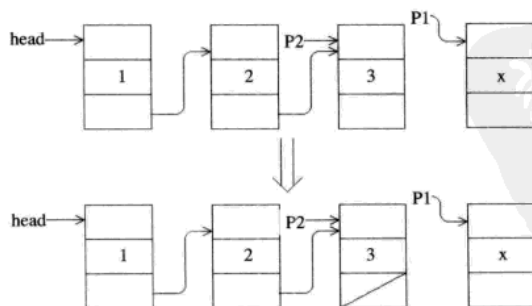


图11.9 链表的建立3



11.3.2 链表的遍历

所谓遍历就是指按一定的规则依次访问结构中的所有数据。遍历链表就是从表头开始，顺序处理链表中的每个结点。链表的遍历算法比较容易：

- 第1步** 以链表的表头结点作为输入。
- 第2步** 设一个指针变量 *p*，先指向第一个结点，并输出该结点中的数据。
- 第3步** 将指针 *p* 向后移动一个结点，再输出结点数据。
- 第4步** 如此继续下去直到链表的尾结点。图 11.10 演示了这个算法的过程。

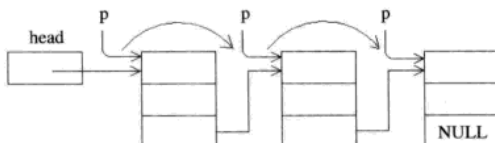


图11.10 链表的遍历

下面我们就来编码实现这个算法。读者还记得上一小节中介绍的链表建立的方法？下面这段示例程序首先根据用户的输入动态地建立一个链表，然后又调用链表遍历函数来将链表中的内容输出。

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE sizeof(struct node)

struct node
{
    int data;
    struct node * next;
};

//新建链表
struct node * initialize(int num)
{
    struct node * head;
    struct node * p1, *p2;
    p1 = p2 = (struct node *)malloc(SIZE);

    printf("\nPlease input the value of the node:\n");
    scanf("%d", &p1->data);
    head = p1;

    while(num > 1)
    {
        p2->next = p1;
        p2 = p1;
        p1 = (struct node *)malloc(SIZE);
        scanf("%d", &p1->data);
        num--;
    }
}
```



```
p2->next = p1;
p2 = p1;
p2->next = NULL;
return head;
}

//遍历函数
void traverse(struct node * head)
{
    struct node * p;
    printf("\nThe records are:\n");
    p = head;
    if(head != NULL)
    {
        do
        {
            printf("%d ", p->data);
            p = p->next;
        }while(p != NULL);
    }
    printf("\n");
}

//主函数
int main()
{
    printf("Please input the number of nodes in list: ");
    int numOfNodes = 0;
    scanf("%d", &numOfNodes);
    struct node * head = initialize(numOfNodes);
    traverse(head);

    return 0;
}
```

请读者完成编码后编译并运行程序。

11.3.3 链表结点的删除

删除链表结点的方法原理已经在介绍链表结构时提过。简单地说,就是从一个动态链表中删除一个结点并非真的是把该结点从内存中销毁,仅仅是修改原链表的链接关系而已。

举个例子来说,如图 11.11 所示的是一辆列车,可以看出它由许多个车厢共同组成。每节车厢都连在前一个车厢后面,除了最后一节车厢以外,每节车厢后面又连着一节车厢。现在如果车厢 B 有了故障,需要将其从这辆列车中删掉,那么实际的做法是怎样的?简单地说,就是让车厢 A 后面不再与 B 相连,而是与车厢 C 相连,也就是解除车厢 B 与车厢 A 和 C 之间的连接关系。

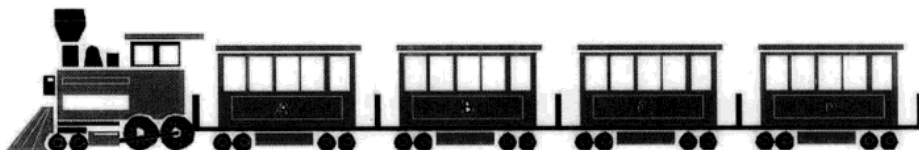



图11.11 火车车厢

在链表中维系结点间链接关系的关键在于每个结点结构体中的 `next` 指针，如果要删除一个结点 `N`，其实就是让 `N` 的前驱结点的 `next` 指针不再指向 `N`，而是改为指向 `N` 结点的后继结点。基于上述原理，下面给出链表结点删除函数的实现代码：

```
struct node * remove(struct node * head, int num)
{
    struct node *p1, *p2;
    //空链表，不做任何处理
    if(head==NULL)
    {
        printf("\nThe list is void!\n");
        return head;
    }

    p1 = head;
    //将 p1 指针不断向后移动，搜索目标结点
    while(num != p1->data && p1->next != NULL)
    {
        p2 = p1;
        p1 = p1->next;
    }
    //如果找到目标结点，则将其删除
    //如果目标结点是头结点，则需要更新头指针
    if(num == p1->data)
    {
        if(p1 != head)
            p2->next = p1->next;
        else
            head = p1->next;
    }
    else
        printf("\nThe node is not found!\n");
    return head;
}
```

其中，参数 `head` 表示链表的头指针，`data` 是要删除的结点中存储的数据值。特别需要提醒读者注意的是，由于头结点可能在函数执行时被删除，所以 `head` 可能会发生变化。因此函数返回一个指向 `struct node` 类型数据的指针，该指针是执行删除操作后链表的新头指针。

 **提示：**读者可以结合前面给出的链表建立函数及链表遍历函数来写一个测试程序，实验一下上述结点删除函数的效果。

11.3.4 链表结点的增加

向链表中插入新结点分多种情况，这主要跟插入结点的位置有关。

1. 向链表的尾部插入新结点

在向链表的表尾插入新结点时，首先把表尾的 NULL 指针指向新加的结点，再把指向原链表的表尾指针向后移动一个结点，也就是让新插入的结点的 next 指针为 NULL 即可。这个过程如图 11.12 所示。

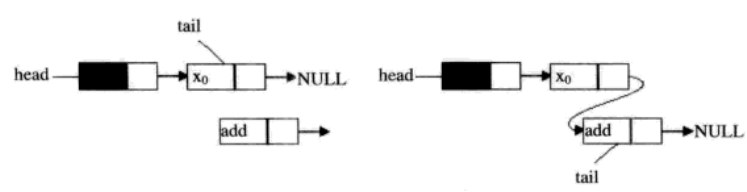


图11.12 在表尾插入结点

2. 向链表中其他位置插入新结点

在向链表中其他位置插入新结点时，表尾指针不变，但要改变链表中相应结点存储的指针，改动过程如图 11.13 所示。可见插入位置的前一个结点的 next 指针将指向新增结点，而新结点的 next 指针则指向插入位置的后一个结点。

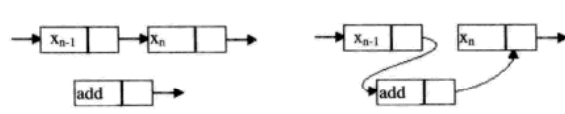


图11.13 在其他位置插入结点

3. 插入一个新的表头结点

插入一个新结点，且该新结点将作为表头结点来使用。这时只需要让头指针指向这个新表头，再让该新结点的 next 指针指向原表头结点即可，整个过程如图 11.14 所示。

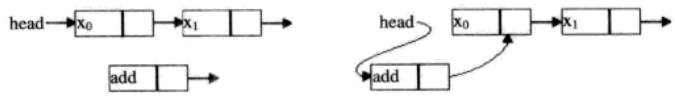


图11.14 在表头位置插入结点

4. 链表结点插入的实现代码

基于上述原理，下面给出链表结点插入函数的实现代码：

```
struct node * insert(struct node * head, int position, int value)
{
    struct node * p1, *p2;
    p1 = head;

    //若链表为空，直接添加
    if(head==NULL)
    {
```



```
        head = (struct node *)malloc(SIZE);
        head->data = value;
        head->next = NULL;
        return head;
    }

    //在尾部插入
    if(position == -1)
    {
        while(p1->next != NULL)
        {
            p2 = p1;
            p1 = p1->next;
        }

        p2 = (struct node *)malloc(SIZE);
        p1->next = p2;
        p2->next = NULL;
        p2->data = value;
    }

    //在头部插入
    if(position == 0)
    {
        p2 = (struct node *)malloc(SIZE);
        p2->data = value;
        p2->next = p1;
        head = p2;
    }

    //在中间的某个位置插入
    if(position > 0)
    {
        while(position > 0 && p1->next != NULL)
        {
            position--;
            p2 = p1;
            p1 = p1->next;
        }
        struct node * newnode = (struct node *)malloc(SIZE);
        p2->next = newnode;
        newnode->data = value;
        newnode->next = p1;
    }

    return head;
}
```

接下来给出一个用于测试上述结点插入函数的主函数，请读者完成编码后运行程序并观察输出结果。

```
int main()
{
    printf("Please input the number of nodes in list: ");
    int numOfNodes = 0;
    scanf("%d", &numOfNodes);
    struct node * head = initialize(numOfNodes);


    head = insert(head, 3, 300);
    head = insert(head, -1, 100);
    head = insert(head, 0, 200);

    traverse(head);

    return 0;
}
```

11.3.5 结点删除函数中的 free() 函数

动态分配的内存必须手动回收, malloc() 函数和 free() 函数必须成对使用。在前面的链表建立和结点增加函数中使用了太多的 malloc() 函数, 而在结点删除函数中, 没有使用一个 free() 函数。这是很可怕的事情, 宝贵的内存被无情地泄漏掉了。

 **注意:** 动态分配的内存必须手动回收。

下面给出修改过的结点删除函数, 其中略去了一些没有发生变化的地方, 请读者注意注释说明的地方。

```
//同前, 此处代码略
//如果找到目标结点, 则将其删除
//如果目标结点是头结点, 则需要更新头指针
if(num == p1->data)
{
    if(p1 != head)
        p2->next = p1->next;
    else
        head = p1->next;
    free(p1); //!!! 注意: 回收内存
}
//同前, 此处代码略
```

11.3.6 链表的应用实例

本小节将举例说明链表的应用。约瑟夫问题是数据结构与算法中的经典问题, 所谓约瑟夫问题, 可简单地表述为: 有 15 个人排成一圈, 并给他们 1~15 的编号。现在从 1 号开始报数, 报数字 4 的人退出队列, 余下的人从退出者下一个位置开始继续刚才的报数, 直到整个队列中只剩一个人为止。请问这个人是几号?

分析: 这个问题乍一看好像很困难。由于每次都有人从队列中退出, 所以队列的长度在不断变化, 这使得整个报数的过程难以预测。但由问题的表述来看, 约瑟夫问题完全可以用



链表来解决。下面就基于前面实现的一些链表操作函数，给出利用链表求解约瑟夫问题的代码清单：

```
int main()
{
    //初始化链表
    struct node * p2 = initialize(15);
    struct node * p1 = p2;

    //让单向链表首尾相接
    while(p1->next != NULL)
    {
        p1 = p1->next;
    }
    p1->next = p2;

    //循环删除结点
    for(int i = 1; i < 15; i++)
    {
        for(int j = 0; j < 3; j++)
        {
            p1 = p2;
            p2 = p2->next;
        }
        p1->next = p2->next;
        free(p2);
        p2 = p1->next;
    }

    //输出结果
    printf("The last number is: ");
    printf("%d\n", p2->data);
    return 0;
}
```

完成编码后，编译并运行程序，输出结果为 13，即最后 13 号会被留下。有兴趣的读者还可以自己试着扩展这个程序，当有 M 个人围成一圈后数到 N 的人退出，求最后剩下的那个人的编号是多少。

11.4 栈

栈是一种非常重要的线性数据结构，它的特点在于最后入栈的数据项也将是最先被处理的数据项。这一小节我们就介绍有关栈的基本知识，并以链表为基础来实现栈结构。

11.4.1 栈定义——散乱的盘子

如果厨房里有一些散乱的盘子，你试图把它们整理在一起。于是你将其中一个摆放在某个平稳的位置上，假设标记这个盘子为 P_0 。然后选取另外一个 P_1 ，并把它摞在 P_0 上。再从散乱的盘子中选出 P_2 ，并把它摞在 P_1 上……如此继续下去直到原来散乱的盘子中最后一个

盘子 P_n 被摞在已经整理好的盘子上为止, 原来散乱的盘子就变成了一叠摞好的盘子, 如图 11.15 所示。

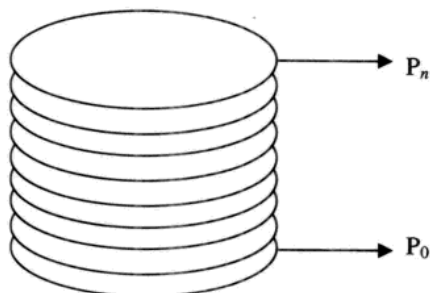


图11.15 摞好的盘子

当你需要使用一个盘子的时候, 想想你会怎么取盘子。显然从中间或者从底部取盘子都不是明智的决定, 因为那样很容易将这叠盘子摔碎。最合理的想法就是总是从最上面开始取出盘子。这个例子是现实世界中一个模拟栈结构的典型例子。

栈是计算机科学中一种重要的数据结构, 它有很多不同的应用。从比较简单的应用到复杂的应用, 随处可见栈的影子, 例如字符串反转、函数活动记录的维护, 以及编译程序中的某些算法实现等都需要以栈为基础。

11.4.2 栈的特点

栈的特点在于仅提供对于最新被加入的数据的访问操作, 而对于其他元素的访问则加以限制。在栈中, 元素的删除和添加只能在表的一端进行。栈中元素的添加和删除操作遵循后进先出 (Last In First Out, LIFO) 的原则, 即后加入栈的数据总是先被访问, 而先加入栈的数据总是后被访问。

换句话说, 下一个能够从栈中被删除的元素永远只能是最近被添加进来的那个。我们总是可以向栈中添加更多的元素, 但每当进行一次添加操作时, 最近被添加进来的元素随即就变成了可以被最先删除的元素。

11.4.3 栈工作原理

先看看栈具体是如何工作的, 假设有如图 11.16 所示的一个栈。使用索引 $\text{tos}(\text{top of stack})$ 来指向栈顶元素。如果 tos 为 -1 , 那么就意味着这个栈是个空栈, 如图 11.16 (a) 所示。

☑ 当向栈中添加一个元素时, 首先把 tos 的值加 1, 然后向此时 tos 指向的位置中添加元素。

☑ 当删除栈中的元素时, 首先把 tos 指向的元素从栈中去掉, 然后令 tos 的值自减 1。

在栈中添加元素的操作通常被称为入栈 (PUSH), 从栈中删除元素的操作通常被称为出栈 (POP), 实现出栈的函数不需要带参数, 因为在某一时刻, 能够被删除的元素至多只有一个, 且它一定位于栈顶。在图 11.16 中首先显示了一个空栈, 然后顺序向栈中添加两个元素 a 与 b , 最后把 b 从栈中删除。

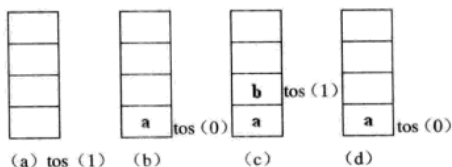


图11.16 栈

从前面的叙述中可以看出, 栈很容易被想象成一种竖立的结构。当向栈中添加一个元素后, 这个元素会把栈中其他的元素“遮盖”起来。这样, 这个新加入的元素便成了栈中唯一可以被访问的元素, 同时也是栈中唯一可以被删除的元素。

11.4.4 栈与链表

1. 栈与链表的不同

在栈中可访问的元素是被限定的, 不可能做到同时访问栈中的所有元素; 而在链表中所有的元素均可被访问。如果想要遍历一个栈, 只能先访问位于栈顶的元素, 访问结束后把这个元素从栈中删除, 然后访问新的栈顶元素, 如此循环下去, 直到遍历结束。由此可以看出, 经过遍历操作后的栈将被清空。由于在栈中访问元素是受到限制的, 所以相对于链表而言, 对栈进行的操作较少。

2. 链表实现栈结构

用链表作为基础可以实现栈结构, 这样的栈又被称作链式栈。在这种情况下, 栈结点的插入和删除都在链表头部执行, 而且拥有栈元素的结点就是链表的结点 (由数据和指向下一结点的指针组成), 如图 11.17 所示。栈 s 可以由只拥有单个指针成员 tos 的一个结构体来表示, tos 是指向栈顶的指针。

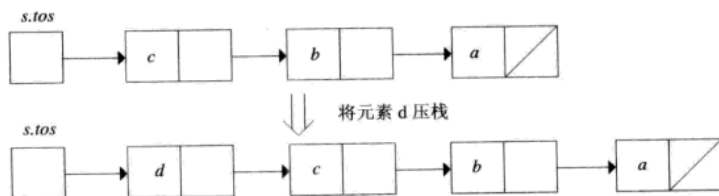


图11.17 链式栈

3. 基于链表的栈结构实现程序

基于上面的思想, 下面给出基于链表的栈结构实现程序, 主函数用于测试出栈、入栈等函数:

```
#include <stdio.h>
#include <stdlib.h>

typedef char stack_element;

// 定义链式栈的结点
```

```
typedef struct stack_node{
    stack_element element;
    struct stack_node * next;
}node;

typedef struct{
    node * tos;
}stack;

//入栈函数
void push(stack *sp, stack_element c)
{
    node *np;

    //创建新结点
    np = (node *)malloc(sizeof(node));
    np->element = c;
    np->next = sp->tos;

    //修改栈顶指针
    sp->tos = np;
}

//出栈函数
stack_element pop(stack * sp)
{
    node * tp;           //将要出栈的结点指针
    stack_element value; //栈顶元素值

    tp = sp->tos;
    value = tp->element;
    sp->tos = tp->next;
    free(tp);
    tp = NULL;

    return value;
}

//遍历输出函数
void show(stack sp)
{
    stack tmpStack = {NULL};
    stack_element tmp;

    while(sp.tos != NULL){
        tmp = pop(&sp);
        printf("%c\n", tmp);
        push(&tmpStack, tmp);
    }
}
```



```
while(tmpStack.tos != NULL)
{
    tmp = pop(&tmpStack);
    push(&sp, tmp);
}

int main()
{
    stack s = {NULL};

    push(&s, 'a');
    push(&s, 'b');
    push(&s, 'c');

    printf("\nThe content of stack: \n");
    show(s);
    printf("\n");

    return 0;
}
```

上述代码中一共提供了 3 个用于对栈结构进行操作的函数。push() 函数用于实现栈结构的元素压栈操作，pop() 函数用于实现栈结构的元素出栈操作，show() 函数实现了对栈中元素进行遍历输出的功能。请读者完成编码后，编译并运行程序观察输出结构。有兴趣的读者也可尝试实现例如清空、判空和元素搜索等其他与栈相关的操作。

11.4.5 栈的应用举例——括号匹配问题

1. 实例分析

现在要求编写一个程序，该程序能够对输入的一段字符串进行括号匹配检查。如果语句中左括号与右括号的数目相等，且能够完整地匹配成对，则表示本语句不存在括号匹配上的错误，此时程序输出“OK!”字样。否则，表示语句中存在语法错误，程序输出“Wrong!”字样。

例如，下面几条语句是存在括号匹配错误的例子：

```
) (
(( ))
```

下面几条语句是括号匹配正确的例子：

```
()
(a) ((b) ((c) (d)))
```

2. 程序实现

利用栈结构，能够非常方便地解决上述问题。下面给出程序实现的代码清单：

```
#include <stdio.h>
#include <stdlib.h>
//注意栈的定义和操作同上节，此处略
int main(){
```

```
printf("请输入括号序列(以0结束):");

    stack s = {NULL};
    char a;
    do{
        scanf("%c",&a);
        switch(a){
            case '(':
                push(&s, a);
                break;
            case ')':
                if(s.tos != NULL){
                    pop(&s);
                    break;
                }
                if(s.tos == NULL){
                    printf("Wrong!\n");
                    return 0;
                }
            }
        }while(a!='0');

        if(s.tos == NULL)
            printf("OK!\n");
        else
            printf("Wrong!\n");

    return 0;
}
```

完成编码后，编译并运行程序即可。

11.5 队 列

队列经常被拿来和栈进行比较。队列也是一种基本的线性数据结构，但操作特性却与栈结构截然相反。在队列中，元素总是在一端添加，而从另一端移出，也就是说最新进入队列的元素将最后被移出。

11.5.1 队列的特点

队列的特点是只允许在结构的一端进行元素添加操作，而元素的删除操作只能在结构的另一端进行。通常将进行元素添加操作的一端称作队列的头，而将进行元素删除操作的一端称作队列的尾。

队列的这种一端删除、一端添加的机制是通过一种被称为先进先出（First In First Out, FIFO）的原则进行来实现的。先进先出意味着最先被加入到队列中的数据将被最先从队列中删除。换句话说，如果想要向队列中添加元素，那么这个操作仅能在队列的头部进行。如果想要从队列中删除一个元素，那么这个操作仅能在队列的尾部进行。



11.5.2 队列定义——排队等待买票的人

在现实生活中存在很多队列的实例。例如在电影院的售票窗口排队等待买票的人，就是通过队列这种数据结构组织在一起的。人们按先来后到的顺序排成一队，最先买到票的人就是最先来的人。当某人买完票，他就会从队列的最前端离开，这就相当于删除操作。而添加的操作却仅能在队列的尾部进行，因此新来的人就只能排在队列的最后。此外还有像公共汽车站人们排队等车、医院中病人们排队候诊等都是现实生活中队列的例子。

11.5.3 队列应用

队列在计算机科学和软件开发中有着很多不同的应用。

1. 队列可以用来提供缓冲区

缓冲区就是一块用来进行临时信息存储的区域，这里的数据稍后就会被处理，因此仅仅是暂时存在缓冲区里。例如网络中的路由器都提供缓冲区，缓冲区将收到数据包组织在一个队列中，以保证最先收到的数据包会被最先发出去。

相信读者一定有过在线收看视频节目的经历，通常与互联网相连接的流媒体程序在播放视频片段时，总是会将收到数据包维护在一个队列中，然后按顺序将其播出，这种缓冲机制可以保证当网络状况不稳定的情况下，视频播放的流畅性。

2. 队列可以维护作业调度及管理输入/输出

操作系统也使用队列来维护作业调度，管理输入/输出，在维护作业调度时，如果有很多等待处理的程序，操作系统就会通过队列来决定下一个将要被运行的程序是哪一个。

3. 队列可以存储待处理打印任务

在打印机中，如果待处理的打印任务很多，那么它们也将被存储在队列中，并由此来决定下一个要执行的打印任务是哪个。这种策略也被称为先来先服务（First Come First Served, FCFS）。

11.5.4 队列与栈的不同

- ☑ 当要向队列中添加新元素时，直接把这个元素添加到队列的末尾即可，这个操作类似于向栈中添加一个新元素。但当从队列中删除一个元素时，则只能删除队列中位于“最前面”的元素，这个操作与从栈中删除一个元素正好相反。
- ☑ 队列需要两个索引值，分别为头（front）和尾（back），而栈只需要一个索引值。当添加元素时，先把新元素添加到此时的 back 指示的位置中（在此之前要判断队列是否已满），然后令索引 back 向后移动一个单位；当删除元素时，直接令 front 向后移动一个单位即可。

11.5.5 队列创建

首先创建一个空队列，然后依次向队列中添加两个元素 a 与 b，如图 11.18 所示。这时 a 存储于队列的头部，是队列中的第一个元素，b 存储于与 a 相邻的下一单元中，是队列中的第二个元素。在向队列中添加元素时，索引 back 会向后移动，而索引 front 不会改变。当从队列中删除元素时，首先删除位于队列头部的 a，随后删除 b。这时，索引 front 会向后移动，

而索引 `back` 不会改变。

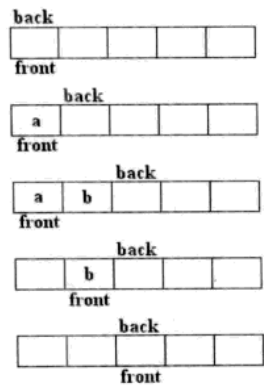


图11.18 队列的基本结构

队列中元素的访问是受限制的，用户只能访问位于队列头部的元素，而无法访问位于其他位置的元素。同栈类似，对队列进行遍历将会把队列置空。对队列进行的基本操作有：向队尾添加一个新元素、删除位于队列头部的元素、获得当前队头元素的值、置空以及判空等。这些操作的实现将在下一小节中给出。

11.5.6 基于链表实现的队列——链式队列

1. 链式队列简述

用链表可以实现队列，这种队列也称为链式队列。链表会在插入和删除元素时进行动态扩展和收缩，为了描述队列的头和尾，我们必须想办法跟踪链表的首个结点和最后一个结点，它们也就是队列的头和尾。图 11.19 给出了一个链式队列的示意图。

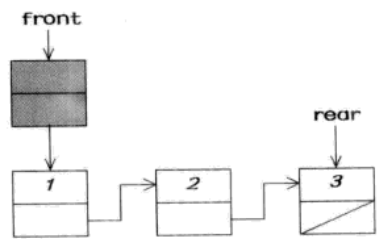



图11.19 链式队列

从上图中可以看出链式队列是基于链表来实现的。但是需要注意的一个问题是在如图 11.19 所示的队列中，队列元素只有 3 个（尽管链表中的结点有 4 个）。事实是这样的，为了方便，首先开辟了一个链结点（图中着灰色的结点），并且在最初的时候让 `front` 指针和 `rear` 指针都指向这个结点。

 **注意：**这个结点并非是实际队列中的一员，它仅仅是为了让 `front` 指针和 `rear` 指针不会指向一个非法地址而设计的。



front 指针所指向的结点的 next 指针真正地指向了队列的头部元素。rear 指针则指向了队列的尾部元素，当链表非空的时候，它的尾部元素就一定存在，而尾部元素结点的特征就是它的 next 指针值为 NULL。

2. 向链表中插入元素

如图 11.20 所示，最开始的时候队列为空，所以 front 指针和 rear 指针都指向初始结点（也就是非队列元素的结点），而且该结点的 next 指针值为 NULL（这也可以用来作为队列判空的依据）。当我们向队列中插入一个新的元素时，front 指针的 next 指针将指向这个新元素，而且 rear 指针也将随之右移。

由于目前队列中只有一个元素，因此这个元素就既是头结点又是尾结点，它的 next 指针值为 NULL。当再次向队列中插入新元素的时候，就如同向链表中插入一个新元素一样，只不过 rear 指针需要随之后移一个位置，因为当队列非空时，rear 指针将永远指向队列的尾部元素。

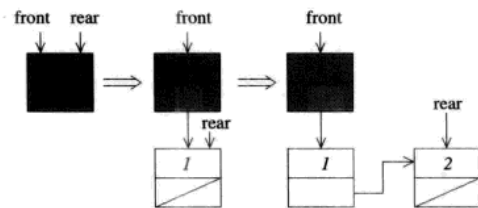


图11.20 向链式队列中添加元素

3. 清空队列和销毁队列的区别

清空队列只是将队列中的元素清零，但是初始结点还将保留，且 front 指针和 rear 指针都指向初始结点。尽管这时队列是一个空队列，但队列仍然存在，因此我们仍然可以向队列中加入新元素。但是如果将队列销毁的话，那么就表示初始结点也会被释放，队列也就不复存在了。有关队列的操作函数的实现方法将在下一小节中给出。

11.5.7 队列的实现

本节将给出队列实现的 C 语言源代码。首先，给出链式队列结构的声明如下，其中 queueNode 是链表结点的声明，LinkQueue 是队列的声明，其中的两个数据项分别是指向队头和队尾的指针：

```
typedef struct queueNode
{
    int data;
    queueNode *next;
}*QueuePtr;

struct LinkQueue
{
    QueuePtr front, rear; //队头、队尾指针
};
```


队列的初始化函数如下，它创建了一个空队列。注意空队列不是空链表，它其实是创建了一个单结点链表。

```
//构造一个空队列
bool initialize(LinkQueue &q)
{
    if(!(q.front=q.rear=(QueuePtr)malloc(sizeof(queueNode))))
    {
        printf("Memory Error!");
        return false;
    }
    q.front->next=NULL;
    return true;
}
```

下面给出的是队列的判空函数，结合上一小节中的描述，读者可以想想是否还有其他的实现方法。

```
//判断队列是否为空
bool isEmpty(LinkQueue q)
{
    if(q.front == q.rear)
        return true;
    else
        return false;
}
```

下面给出队列的其他操作的实现方法，包括新增元素、遍历队列、获取队列长度和删除队列元素等，限于篇幅这里就不一一说明了，请读者留意注释部分对于函数所实现功能的说明。

```
//插入一个新的队尾元素
void add(LinkQueue &q, int value)
{
    QueuePtr p = (QueuePtr)malloc(sizeof(queueNode));
    if(!p)
    {
        printf("Memory Error!");
        return;
    }
    p->data = value;
    p->next = NULL;

    q.rear->next = p;
    q.rear = p;
}

//遍历输出队列
void traverse(LinkQueue q)
{
    QueuePtr p;
    p = q.front->next;
```



```
printf("The values in the queue are:\n");
while(p)
{
    printf("%d ",p->data);
    p = p->next;
}
printf("\n");
}
```

//求队列的长度

```
int getLength(struct LinkQueue q)
```

```
{
    int i = 0;
    QueuePtr p;
    p = q.front;
    while(q.rear != p)
    {
        i++;
        p = p->next;
    }
    return i;
}
```

//读取队列首元素

//若队列不空, 则返回 True, 否则返回 False

```
bool getHead(LinkQueue q, int * value)
```

```
{
    if(q.front == q.rear)
        return false;
    * value = q.front->next->data;
    return true;
}
```

// 若队列不空, 删除队头元素

```
bool remove(LinkQueue &q)
```

```
{
    QueuePtr p;
    if(q.front==q.rear)
        return false;
    p=q.front->next;

    q.front->next=p->next;
    if(q.rear==p)
        q.rear=q.front;
    free(p);
    return true;
}
```

下面给出队列销毁的函数实现, 有兴趣的读者也可以尝试着写一下队列清空的函数, 注意对比它们二者实现上的不同。

```
//销毁队列
void destroy(LinkQueue &q)
{
    while(q.front)
    {
        q.rear=q.front->next;
        free(q.front);
        q.front=q.rear;
    }
}
```

最后给出一个用于测试上述队列结构和操作的主函数，代码清单如下：

```
#include<malloc.h>
#include<stdio.h>

//队列的定义及实现同上，此处略

int main()
{
    LinkQueue queue;
    initialize(queue);
    isEmpty(queue)==true ?
    printf("Empty Queue!\n") : printf("Not Empty Queue!\n");

    int a[8] = {1, 2, 3, 4, 5, 6, 7, 8};

    int i = 0;
    for(; i<8; i++){
        add(queue, a[i]);
    }

    traverse(queue);

    printf("The length of the queue is: %d\n", getLength(queue));

    int value;
    if(getHead(queue, &value)==true)
        printf("The head value is: %d\n", value);

    remove(queue);
    printf("\nAfter delete the head of the queue...\n");
    traverse(queue);

    destroy(queue);
    printf("\nDestroy the queue...\n");
    printf("queue.front=%d queue.rear=%d\n",queue.front, queue.rear);

    return 0;
}
```

请读者完成编码后，运行程序并观察结果。

附录 I C 语言运算符及其优先级汇总表

优先级	运 算 符	解 释	结合方式
1	() [] -> .	括号 (函数等) 数组 指向结构体成员 结构体成员	由左向右
2	! ~ ++ -- + - * & (类型) sizeof	否定 按位否定 增量 减量 正号 负号 间接 取地址 类型转换 求长度	由右向左
3	* / %	乘 除 取余	由左向右
4	+ -	加 减	由左向右
5	<< >>	左移 右移	由左向右
6	<, <=, >, >=, >	小于、小于等于、大于等于、大于	由左向右
7	== !=	等于 不等于	由左向右
8	&	按位与	由左向右
9	^	按位异或	由左向右
10		按位或	由左向右
11	&&	逻辑与	由左向右
12		逻辑或	由左向右
13	?:	条件	由右向左
14	=, +=, -=, *=, /=, &=, ^=, =, <<=, >>=	各种赋值	由右向左
15	,	逗号 (顺序)	由左向右

说明:

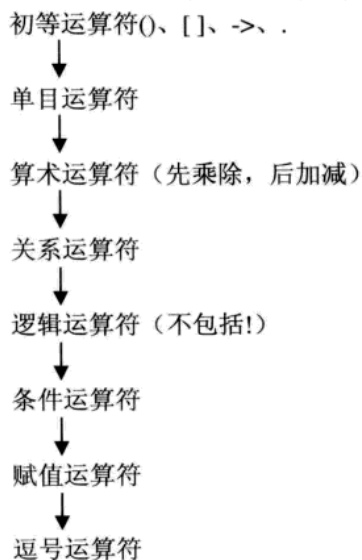
(1) 同一优先级的运算符优先级别相同, 运算次序由结合方向决定。例如 “*” 与 “/”

运算符具有相同的优先级别，其结合方向为自左至右，因此 $4*3/2$ 的运算次序是先乘后除。

“-”和“++”运算符为同一优先级，结合方向为自右至左，因此 $-i++$ 相当于 $-(i++)$ 。

(2) 不同的运算符要求有不同的运算对象个数，如“*”（乘）和“/”（除）运算符为双目运算符，要求在运算符两侧各有一个运算对象（如 $1*2$ 、 $8/2$ 等）。而“-”和“+”（正号）运算符是一元运算符，只能在运算符的一侧出现一个运算对象（如 $x++$ 、 $-i$ 、 $*p$ 等）。条件运算符是 C 语言中唯一的一个三目运算符。

(3) 从上述表中可以大致归纳出各类运算符的优先级：



以上运算符的优先级别由上到下递减。初等运算符优先级最高，逗号运算符优先级最低。位运算符的优先级比较分散，有的在算术运算符之前（如~），有的在关系运算符之前（如<<和>>），有的则在关系运算符之后（如&、^、|）。为了容易记忆和表述清晰，使用位运算符时可加圆括号。



附录 II 标准 ASCII 码字符集

标准 ASCII 码共有 128 个字符，其编码从 0~127。表 I1 列出常用字符的 ASCII 编码值。

表I1 标准ASCII表

ASCII 值	字 符	ASCII 值	字 符	ASCII 值	字 符	ASCII 值	字 符
0	NUL	32	(Space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(72	H	104	h
9	HT	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	X	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	/	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

其中, ASCII 码从 0~31 的字符的含义如表 I2 所示。

表 I2 0~31 的字符的含义

ASCII 值	字 符	含 义	ASCII 值	字 符	含 义
0	NUL	空字符	16	DLE	数据链路转义
1	SOH	标题开始	17	DC1	设备控制 1
2	STX	正文开始	18	DC2	设备控制 2
3	ETX	正文结束	19	DC3	设备控制 3
4	EOT	传输结束	20	DC4	设备控制 4
5	ENQ	请求	21	NAK	拒绝接收
6	ACK	收到通知	22	SYN	同步空闲
7	BEL	响铃	23	ETB	传输块结束
8	BS	退格	24	CAN	取消
9	HT	水平制表符	25	EM	介质中断
10	LF	换行	26	SUB	减
11	VT	垂直制表符	27	ESC	溢出
12	FF	换页	28	FS	文件分隔符
13	CR	回车	29	GS	分组符
14	SO	移位输出	30	RS	记录分隔符
15	SI	启用移位输入	31	US	单元分隔符

